learnbyexample

GNU SED

/awesome stream editor/

Search & Replace

Transform

Filter

Sundeep Agarwal

Table of contents

Preface **5**
- Prerequisites . . . 5
- Conventions . . . 5
- Acknowledgements . . . 6
- Feedback and Errata . . . 6
- Author info . . . 6
- License . . . 6
- Book version . . . 7

Introduction **8**
- Installation . . . 8
- Documentation and options overview . . . 8
- Editing standard input . . . 10
- Editing file input . . . 11
- Cheatsheet and summary . . . 11
- Exercises . . . 12

In-place file editing **13**
- With backup . . . 13
- Without backup . . . 13
- Multiple files . . . 14
- Prefix backup name . . . 14
- Place backups in different directory . . . 15
- Cheatsheet and summary . . . 15
- Exercises . . . 15

Selective editing **17**
- Conditional execution . . . 17
- Delete command . . . 17
- Print command . . . 18
- Quit commands . . . 19
- Multiple commands . . . 20
- Line addressing . . . 21
- Print only line number . . . 22
- Address range . . . 23
- Relative addressing . . . 24
- n and N commands . . . 26
- Cheatsheet and summary . . . 27
- Exercises . . . 28

BRE/ERE Regular Expressions **31**
- Line Anchors . . . 31
- Word Anchors . . . 32
- Alternation . . . 33
- Grouping . . . 34
- Matching the metacharacters . . . 35
- Using different delimiters . . . 36
- The dot meta character . . . 36
- Quantifiers . . . 37

Longest match wins . . . 39
Character classes . . . 40
Escape sequences . . . 43
Backreferences . . . 44
Known Bugs . . . 46
Cheatsheet and summary . . . 47
Exercises . . . 48

Flags **51**
Case insensitive matching . . . 51
Changing case in replacement section . . . 51
Global replace . . . 52
Replace specific occurrences . . . 53
Print flag . . . 54
Write to a file . . . 54
Executing external commands . . . 55
Multiline mode . . . 57
Cheatsheet and summary . . . 58
Exercises . . . 59

Shell substitutions **62**
Variable substitution . . . 62
Escaping metacharacters . . . 63
Command substitution . . . 64
Cheatsheet and summary . . . 65
Exercises . . . 66

z, s and f command line options **67**
NUL separated lines . . . 67
Separate files . . . 68
File as source of sed commands . . . 68
Cheatsheet and summary . . . 70
Exercises . . . 70

append, change, insert **72**
Basic usage . . . 72
Escape sequences . . . 73
Multiple commands . . . 74
Shell substitution . . . 75
Cheatsheet and summary . . . 75
Exercises . . . 76

Adding content from file **77**
r for entire file . . . 77
Using e and cat command . . . 79
R for line by line . . . 79
Cheatsheet and summary . . . 80
Exercises . . . 80

Control structures **82**
Branch commands . . . 82

if-then-else . . . 82
loop . . . 83
Cheatsheet and summary . . . 85
Exercises . . . 85

Processing lines bounded by distinct markers **87**
Uniform markers . . . 87
Extracting first or last group . . . 89
Broken groups . . . 90
Summary . . . 90
Exercises . . . 91

Gotchas and Tricks **92**

Further Reading **100**

Preface

You are likely to be familiar with the ”Find and Replace” dialog box from a text editor, word processor, IDE, etc to search for something and replace it with something else. `sed` is a command line tool that is similar, but much more versatile and feature-rich. Some of the GUI applications may also support **regular expressions**, a feature which helps to precisely define a matching criteria. You could consider regular expressions as a mini-programming language in itself, designed to solve various text processing needs.

The book heavily leans on examples to present options and features of `sed` one by one. Regular expressions will also be discussed in detail. However, commands to manipulate data buffers and multiline techniques will be discussed only briefly and some commands are skipped entirely.

It is recommended that you manually type each example and experiment with them. Understanding both the nature of sample input string and the output produced is essential. As an analogy, consider learning to drive a bike or a car — no matter how much you read about them or listen to explanations, you need to practice a lot and infer your own conclusions. Should you feel that copy-paste is ideal for you, code snippets are available chapter wise on GitHub.

Prerequisites

Prior experience working with command line and `bash` shell, should know concepts like file redirection, command pipeline and so on. Knowing basics of `grep` will also help in understanding filtering features of `sed` .

If you are new to the world of command line, check out ryanstutorials or my GitHub repository on Linux Command Line before starting this book.

My Command Line Text Processing repository includes a chapter on `GNU sed` which has been edited and expanded to create this book.

Conventions

- The examples presented here have been tested on `GNU bash` shell with **GNU sed 4.8** and may include features not available in earlier versions
- Code snippets shown are copy pasted from `bash` shell and modified for presentation purposes. Some commands are preceded by comments to provide context and explanations. Blank lines to improve readability, only `real` time shown for speed comparisons, output skipped for commands like `wget` and so on
- Unless otherwise noted, all examples and explanations are meant for *ASCII* characters only
- `sed` would mean `GNU sed` , `grep` would mean `GNU grep` and so on unless otherwise specified
- External links are provided for further reading throughout the book. Not necessary to immediately visit them. They have been chosen with care and would help, especially during rereads
- The learn_gnused repo has all the files used in examples and exercises and other details related to the book. Click the **Clone or download** button to get the files

Acknowledgements

- GNU sed documentation — manual and examples
- stackoverflow and unix.stackexchange — for getting answers to pertinent questions on `bash` , `sed` and other commands
- tex.stackexchange — for help on `pandoc` and `tex` related questions
- Cover image
    - LibreOffice Draw
- softwareengineering.stackexchange and skolakoda for programming quotes
- Warning and Info icons by Amada44 under public domain
- arifmahmudrana for spotting an ambiguous explanation

Special thanks to all my friends and online acquaintances for their help, support and encouragement, especially during difficult times.

Feedback and Errata

I would highly appreciate if you'd let me know how you felt about this book, it would help to improve this book as well as my future attempts. Also, please do let me know if you spot any error or typo.

Issue Manager: https://github.com/learnbyexample/learn_gnused/issues

E-mail: learnbyexample.net@gmail.com

Twitter: https://twitter.com/learn_byexample

Author info

Sundeep Agarwal is a freelance trainer, author and mentor. His previous experience includes working as a Design Engineer at Analog Devices for more than 5 years. You can find his other works, primarily focused on Linux command line, text processing, scripting languages and curated lists, at https://github.com/learnbyexample. He has also been a technical reviewer for Command Line Fundamentals book and video course published by Packt.

List of books: https://learnbyexample.github.io/books/

License

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License

Code snippets are available under MIT License

Resources mentioned in Acknowledgements section above are available under original licenses.

Book version

1.3

See Version_changes.md to track changes across book versions.

Introduction

The command name `sed` is derived from **s**tream **ed**itor. Here, stream refers to data being passed via shell pipes. Thus, the command's primary functionality is to act as a text editor for **stdin** data with **stdout** as output target. Over the years, functionality was added to edit file input and save the changes back to the same file.

This chapter will cover how to install/upgrade `sed` followed by details related to documentation. Then, you'll get an introduction to **substitute** command, which is the most commonly used `sed` feature. The chapters to follow will add more details to the substitute command, discuss other commands and command line options. Cheatsheet, summary and exercises are also included at the end of these chapters.

Installation

If you are on a Unix like system, you are most likely to already have some version of `sed` installed. This book is primarily for `GNU sed` . As there are syntax and feature differences between various implementations, please make sure to follow along with what is presented here. `GNU sed` is part of text creation and manipulation commands provided by `GNU` and comes by default on GNU/Linux. To install newer or particular version, visit gnu: software. Check release notes for an overview of changes between versions. See also bug list.

```
$ # use a dir, say ~/Downloads/sed_install before following the steps below
$ wget https://ftp.gnu.org/gnu/sed/sed-4.8.tar.xz
$ tar -Jxf sed-4.8.tar.xz
$ cd sed-4.8/
$ ./configure
$ make
$ sudo make install

$ type -a sed
sed is /usr/local/bin/sed
sed is /bin/sed
$ sed --version | head -n1
sed (GNU sed) 4.8
```

If you are not using a Linux distribution, you may be able to access `GNU sed` using below options:

- git-bash
- WSL
- brew

Documentation and options overview

It is always a good idea to know where to find the documentation. From command line, you can use `man sed` for a short manual and `info sed` for full documentation. For a better interface, visit online gnu sed manual.

```
$ man sed
SED(1)                          User Commands                          SED(1)

NAME
       sed - stream editor for filtering and transforming text

SYNOPSIS
       sed [OPTION]... {script-only-if-no-other-script} [input-file]...

DESCRIPTION
       Sed  is  a  stream  editor.  A stream editor is used to perform basic
       text transformations on an input stream (a file or input from a pipe-
       line).  While  in  some  ways  similar  to  an  editor which permits
       scripted edits (such as ed), sed works by making only one  pass  over
       the  input(s),  and  is consequently more efficient.  But it is sed's
       ability to filter text in a pipeline which particularly distinguishes
       it from other types of editors.
```

For a quick overview of all the available options, use `sed --help` from the command line. Most of them will be explained in the coming chapters.

```
$ # only partial output shown here
$ sed --help
  -n, --quiet, --silent
                 suppress automatic printing of pattern space
      --debug
                 annotate program execution
  -e script, --expression=script
                 add the script to the commands to be executed
  -f script-file, --file=script-file
                 add the contents of script-file to the commands to be executed
  --follow-symlinks
                 follow symlinks when processing in place
  -i[SUFFIX], --in-place[=SUFFIX]
                 edit files in place (makes backup if SUFFIX supplied)
  -l N, --line-length=N
                 specify the desired line-wrap length for the 'l' command
  --posix
                 disable all GNU extensions.
  -E, -r, --regexp-extended
                 use extended regular expressions in the script
                 (for portability use POSIX -E).
  -s, --separate
                 consider files as separate rather than as a single,
                 continuous long stream.
      --sandbox
                 operate in sandbox mode (disable e/r/w commands).
  -u, --unbuffered
                 load minimal amounts of data from the input files and flush
```

```
                  the output buffers more often
  -z, --null-data
                  separate lines by NUL characters
      --help      display this help and exit
      --version   output version information and exit

If no -e, --expression, -f, or --file option is given, then the first
non-option argument is taken as the sed script to interpret.  All
remaining arguments are names of input files; if no input files are
specified, then the standard input is read.
```

Editing standard input

`sed` has various commands to manipulate text input. **substitute** command is most commonly used, which will be briefly discussed in this chapter. It is used to replace matching text with something else. The syntax is `s/REGEXP/REPLACEMENT/FLAGS` where

- `s` stands for **substitute** command
- `/` is an idiomatic delimiter character to separate various portions of the command
- `REGEXP` stands for **regular expression**
- `REPLACEMENT` specifies the replacement string
- `FLAGS` are options to change default behavior of the command

For now, it is enough to know that `s` command is used for search and replace operation.

```
$ # sample command output for stream editing
$ printf '1,2,3,4\na,b,c,d\n'
1,2,3,4
a,b,c,d

$ # for each input line, change only first ',' to '-'
$ printf '1,2,3,4\na,b,c,d\n' | sed 's/,/-/'
1-2,3,4
a-b,c,d

$ # change all matches by adding 'g' flag
$ printf '1,2,3,4\na,b,c,d\n' | sed 's/,/-/g'
1-2-3-4
a-b-c-d
```

Here sample input is created using `printf` command to showcase stream editing. By default, `sed` processes input line by line. To determine a line, `sed` uses the `\n` newline character. The first `sed` command replaces only the first occurrence of `,` with `-`. The second command replaces all occurrences as `g` flag is also used (`g` stands for `global`).

If you have a file with a different line ending style, you'll need to preprocess it first. For example, a text file downloaded from internet or a file originating from Windows OS would typically have lines ending with `\r\n` (carriage return + line feed). Modern text editors, IDEs and word processors can handle both styles easily. But every character matters when it comes to command line text processing. See stackoverflow: Why does my tool output overwrite itself and how do I fix it? for a detailed discussion and mitigation methods.

As a good practice, always use single quotes around the script to prevent shell interpretation. Other variations will be discussed later.

Editing file input

Although `sed` derives its name from *stream editing*, it is common to use `sed` for file editing. To do so, append one or more input filenames to the command. You can also specify `stdin` as a source by using `-` as filename. By default, output will go to `stdout` and the input files will not be modified. In-place file editing chapter will discuss how to apply the changes to source file.

Sample input files used in examples are available from learn_gnused repo.

```
$ cat greeting.txt
Hi there
Have a nice day

$ # for each line, change first occurrence of 'day' with 'weekend'
$ sed 's/day/weekend/' greeting.txt
Hi there
Have a nice weekend

$ # change all 'e' to 'E' and save changed text to another file
$ sed 's/e/E/g' greeting.txt > out.txt
$ cat out.txt
Hi thErE
HavE a nicE day
```

In the previous section examples, all input lines had matched the search expression. The first `sed` command here searched for `day`, which did not match the first line of `greeting.txt` file input. By default, even if a line didn't satisfy the search expression, it will be part of the output. You'll see how to get only the modified lines in Print command section.

Cheatsheet and summary

Note	Description
`man sed`	brief manual
`sed --help`	brief description of all the command line options
`info sed`	comprehensive manual
online gnu sed manual	well formatted, easier to read and navigate
`s/REGEXP/REPLACEMENT/FLAGS`	syntax for substitute command
`sed 's/,/-/'`	replace first `,` with `-`
`sed 's/,/-/g'`	replace all `,` with `-`

This introductory chapter covered installation process, documentation and how to search and replace basic text using `sed` from the command line. In coming chapters, you'll learn many more commands and features that make `sed` an important tool when it comes to command line text processing. One such feature is editing files in-place, which will be discussed in the next chapter.

Exercises

Exercise related files are available from exercises folder of learn_gnused repo.

a) Replace `5` with `five` for the given stdin source.

```
$ echo 'They ate 5 apples' | sed ##### add your solution here
They ate five apples
```

b) Replace all occurrences of `0xA0` with `0x50` and `0xFF` with `0x7F` for the given input file.

```
$ cat hex.txt
start address: 0xA0, func1 address: 0xA0
end address: 0xFF, func2 address: 0xB0

$ sed ##### add your solution here
start address: 0x50, func1 address: 0x50
end address: 0x7F, func2 address: 0xB0
```

c) The substitute command searches and replaces sequences of characters. When you need to map one or more characters with another set of corresponding characters, you can use the `y` command. Quoting from the manual:

> **y/src/dst/** Transliterate any characters in the pattern space which match any of the source-chars with the corresponding character in dest-chars.

Use the `y` command to transform the given input string to get the output string as shown below.

```
$ echo 'goal new user sit eat dinner' | sed ##### add your solution here
gOAl nEw UsEr sIt EAt dInnEr
```

In-place file editing

In the examples presented in previous chapter, the output from `sed` was displayed on the terminal or redirected to another file. This chapter will discuss how to write back the changes to the input file(s) itself using the `-i` command line option. This option can be configured to make changes to the input file(s) with or without creating a backup of original contents. When backups are needed, the original filename can get a prefix or a suffix or both. And the backups can be placed in the same directory or some other directory as needed.

With backup

When an extension is provided as an argument to `-i` option, the original contents of the input file gets preserved as per the extension given. For example, if the input file is `ip.txt` and `-i.orig` is used, the backup file will be named as `ip.txt.orig`

```
$ cat colors.txt
deep blue
light orange
blue delight

$ # no output on terminal as -i option is used
$ # space is NOT allowed between -i and extension
$ sed -i.bkp 's/blue/green/' colors.txt
$ # output from sed is written back to 'colors.txt'
$ cat colors.txt
deep green
light orange
green delight

$ # original file is preserved in 'colors.txt.bkp'
$ cat colors.txt.bkp
deep blue
light orange
blue delight
```

Without backup

Sometimes backups are not desirable. Using `-i` option on its own will prevent creating backups. Be careful though, as changes made cannot be undone. In such cases, test the command with sample input before using `-i` option on actual file. You could also use the option with backup, compare the differences with a `diff` program and then delete the backup.

```
$ cat fruits.txt
banana
papaya
mango

$ sed -i 's/an/AN/g' fruits.txt
```

```
$ cat fruits.txt
bANANa
papaya
mANgo
```

Multiple files

Multiple input files are treated individually and the changes are written back to respective files.

```
$ cat f1.txt
have a nice day
bad morning
what a pleasant evening
$ cat f2.txt
worse than ever
too bad

$ sed -i.bkp 's/bad/good/' f1.txt f2.txt
$ ls f?.*
f1.txt  f1.txt.bkp  f2.txt  f2.txt.bkp

$ cat f1.txt
have a nice day
good morning
what a pleasant evening
$ cat f2.txt
worse than ever
too good
```

Prefix backup name

A `*` character in the argument to `-i` option is special. It will get replaced with input filename. This is helpful if you need to use a prefix instead of suffix for the backup filename. Or any other combination that may be needed.

```
$ ls *colors*
colors.txt  colors.txt.bkp

$ # single quotes is used here as * is a special shell character
$ sed -i'bkp.*' 's/green/yellow/' colors.txt
$ ls *colors*
bkp.colors.txt  colors.txt  colors.txt.bkp
```

Place backups in different directory

The `*` trick can also be used to place the backups in another directory instead of the parent directory of input files. The backup directory should already exist for this to work.

```
$ mkdir backups
$ sed -i'backups/*' 's/good/nice/' f1.txt f2.txt
$ ls backups/
f1.txt  f2.txt
```

Cheatsheet and summary

Note	Description
`-i`	after processing, write back changes to input file itself changes made cannot be undone, so use this option with caution
`-i.bkp`	in addition to in-place editing, preserve original contents to a file whose name is derived from input filename and `.bkp` as a suffix
`-i'bkp.*'`	`*` here gets replaced with input filename thus providing a way to add a prefix instead of a suffix
`-i'backups/*'`	this will place the back up copy in a different existing directory instead of source directory

This chapter discussed about the `-i` option which is useful when you need to edit a file in-place. This is particularly useful in automation scripts. But, do ensure that you have tested the `sed` command before applying to actual files if you need to use this option without creating backups. In the next chapter, you'll learn filtering features of `sed` and how that helps to apply commands to only certain input lines instead of all the lines.

Exercises

a) For the input file `text.txt`, replace all occurrences of `in` with `an` and write back the changes to `text.txt` itself. The original contents should get saved to `text.txt.orig`

```
$ cat text.txt
can ran want plant
tin fin fit mine line
$ sed ##### add your solution here

$ cat text.txt
can ran want plant
tan fan fit mane lane
$ cat text.txt.orig
can ran want plant
tin fin fit mine line
```

b) For the input file `text.txt`, replace all occurrences of `an` with `in` and write back the changes to `text.txt` itself. Do not create backups for this exercise. Note that you should

have solved the previous exercise before starting this one.

```
$ cat text.txt
can ran want plant
tan fan fit mane lane
$ sed ##### add your solution here

$ cat text.txt
cin rin wint plint
tin fin fit mine line
$ diff text.txt text.txt.orig
1c1
< cin rin wint plint
---
> can ran want plant
```

c) For the input file `copyright.txt`, replace `copyright: 2018` with `copyright: 2019` and write back the changes to `copyright.txt` itself. The original contents should get saved to `2018_copyright.txt.bkp`

```
$ cat copyright.txt
bla bla 2015 bla
blah 2018 blah
bla bla bla
copyright: 2018
$ sed ##### add your solution here

$ cat copyright.txt
bla bla 2015 bla
blah 2018 blah
bla bla bla
copyright: 2019
$ cat 2018_copyright.txt.bkp
bla bla 2015 bla
blah 2018 blah
bla bla bla
copyright: 2018
```

d) In the code sample shown below, two files are created by redirecting output of `echo` command. Then a `sed` command is used to edit `b1.txt` in-place as well as create a backup named `bkp.b1.txt`. Will the `sed` command work as expected? If not, why?

```
$ echo '2 apples' > b1.txt
$ echo '5 bananas' > -ibkp.txt
$ sed -ibkp.* 's/2/two/' b1.txt
```

Selective editing

By default, `sed` acts on entire file. Many a times, you only want to act upon specific portions of file. To that end, `sed` has features to filter lines, similar to tools like `grep` , `head` and `tail` . `sed` can replicate most of `grep` 's filtering features without too much fuss. And has features like line number based filtering, selecting lines between two patterns, relative addressing, etc which isn't possible with `grep` . If you are familiar with functional programming, you would have come across **map, filter, reduce** paradigm. A typical task with `sed` involves filtering subset of input and then modifying (mapping) them. Sometimes, the subset is entire input file, as seen in the examples of previous chapters.

A tool optimized for a particular functionality should be preferred where possible. `grep` , `head` and `tail` would be better performance wise compared to `sed` for equivalent line filtering solutions.

For some of the examples, equivalent commands will be shown as comments for learning purposes.

Conditional execution

As seen earlier, the syntax for substitute command is `s/REGEXP/REPLACEMENT/FLAGS` . The `/REGEXP/FLAGS` portion can be used as a conditional expression to allow commands to execute only for the lines matching the pattern.

```
$ # change commas to hyphens only if the input line contains '2'
$ # space between the filter and command is optional
$ printf '1,2,3,4\na,b,c,d\n' | sed '/2/ s/,/-/g'
1-2-3-4
a,b,c,d
```

Use `/REGEXP/FLAGS!` to act upon lines other than the matching ones.

```
$ # change commas to hyphens if the input line does NOT contain '2'
$ # space around ! is optional
$ printf '1,2,3,4\na,b,c,d\n' | sed '/2/! s/,/-/g'
1,2,3,4
a-b-c-d
```

`/REGEXP/` is one of the ways to define a filter in `sed` , termed as **address** in the manual. Others will be covered in sections to come in this chapter.

Delete command

To **d**elete the filtered lines, use the `d` command. Recall that all input lines are printed by default.

```
$ # same as: grep -v 'at'
$ printf 'sea\neat\ndrop\n' | sed '/at/d'
```

```
sea
drop
```

To get the default `grep` filtering, use `!d` combination. Sometimes, negative logic can get confusing to use. It boils down to personal preference, similar to choosing between `if` and `unless` conditionals in programming languages.

```
$ # same as: grep 'at'
$ printf 'sea\neat\ndrop\n' | sed '/at/!d'
eat
```

Using an **address** is optional. So, for example, `sed '!d' file` would be equivalent to `cat file` command.

Print command

To **p**rint the filtered lines, use the `p` command. But, recall that all input lines are printed by default. So, this command is typically used in combination with `-n` command line option, which would turn off the default printing.

```
$ cat programming_quotes.txt
Debugging is twice as hard as writing the code in the first place.
Therefore, if you write the code as cleverly as possible, you are,
by definition, not smart enough to debug it by Brian W. Kernighan

Some people, when confronted with a problem, think - I know, I will
use regular expressions. Now they have two problems by Jamie Zawinski

A language that does not affect the way you think about programming,
is not worth knowing by Alan Perlis

There are 2 hard problems in computer science: cache invalidation,
naming things, and off-by-1 errors by Leon Bambrick

$ # same as: grep 'twice' programming_quotes.txt
$ sed -n '/twice/p' programming_quotes.txt
Debugging is twice as hard as writing the code in the first place.
$ # same as: grep 'e th' programming_quotes.txt
$ sed -n '/e th/p' programming_quotes.txt
Therefore, if you write the code as cleverly as possible, you are,
A language that does not affect the way you think about programming,
```

The substitute command provides `p` as a flag. In such a case, the modified line would be printed only if the substitution succeeded.

```
$ # same as: grep '1' programming_quotes.txt | sed 's/1/one/g'
$ sed -n 's/1/one/gp' programming_quotes.txt
naming things, and off-by-one errors by Leon Bambrick
```

```
$ # filter + substitution + p combination
$ # same as: grep 'not' programming_quotes.txt | sed 's/in/**/g'
$ sed -n '/not/ s/in/**/gp' programming_quotes.txt
by def**ition, not smart enough to debug it by Brian W. Kernighan
A language that does not affect the way you th**k about programm**g,
is not worth know**g by Alan Perlis
```

Using `!p` with `-n` option will be equivalent to using `d` command.

```
$ # same as: sed '/at/d'
$ printf 'sea\neat\ndrop\n' | sed -n '/at/!p'
sea
drop
```

Here's an example of using `p` command without the `-n` option.

```
$ # duplicate every line
$ seq 2 | sed 'p'
1
1
2
2
```

Quit commands

Using `q` command will exit `sed` immediately, without any further processing.

```
$ # quits after an input line containing 'if' is found
$ sed '/if/q' programming_quotes.txt
Debugging is twice as hard as writing the code in the first place.
Therefore, if you write the code as cleverly as possible, you are,
```

`Q` command is similar to `q` but won't print the matching line.

```
$ # matching line won't be printed
$ sed '/if/Q' programming_quotes.txt
Debugging is twice as hard as writing the code in the first place.
```

Use `tac` to get all lines starting from last occurrence of the search string with respect to entire file content.

```
$ tac programming_quotes.txt | sed '/not/q' | tac
is not worth knowing by Alan Perlis

There are 2 hard problems in computer science: cache invalidation,
naming things, and off-by-1 errors by Leon Bambrick
```

You can optionally provide an exit status (from `0` to `255`) along with the quit commands.

```
$ printf 'sea\neat\ndrop\n' | sed '/at/q2'
sea
eat
$ echo $?
```

```
2

$ printf 'sea\neat\ndrop\n' | sed '/at/Q3'
sea
$ echo $?
3
```

Be careful if you want to use `q` or `Q` commands with multiple files, as `sed` will stop even if there are other files to process. You could use a mixed address range as a workaround. See also unix.stackexchange: applying q to multiple files.

Multiple commands

Commands seen so far can be specified more than once by separating them using `;` or using the `-e` command line option. See sed manual: Multiple commands syntax for more details.

```
$ # print all input lines as well as modified lines
$ printf 'sea\neat\ndrop\n' | sed -n -e 'p' -e 's/at/AT/p'
sea
eat
eAT
drop

$ # equivalent command to above example using ; instead of -e
$ # space around ; is optional
$ printf 'sea\neat\ndrop\n' | sed -n 'p; s/at/AT/p'
sea
eat
eAT
drop
```

Another way is to separate the commands using a literal newline character. If more than 2-3 lines are needed, it is better to use a sed script instead.

```
$ # here, each command is separated by literal newline character
$ # > at start of line indicates continuation of multiline shell command
$ sed -n '
> /not/ s/in/**/gp
> s/1/one/gp
> s/2/two/gp
> ' programming_quotes.txt
by def**ition, not smart enough to debug it by Brian W. Kernighan
A language that does not affect the way you th**k about programm**g,
is not worth know**g by Alan Perlis
There are two hard problems in computer science: cache invalidation,
naming things, and off-by-one errors by Leon Bambrick
```

Do not use multiple commands to construct conditional OR of multiple search strings, as you might get lines duplicated in the output. For example, check what output you get for `sed -ne '/use/p' -e '/two/p' programming_quotes.txt` command. You can use regular expression feature alternation for such cases.

To execute multiple commands for a common filter, use `{}` to group the commands. You can also nest them if needed.

```
$ # same as: sed -n 'p; s/at/AT/p'
$ printf 'sea\neat\ndrop\n' | sed '/at/{p; s/at/AT/}'
sea
eat
eAT
drop

$ # spaces around {} is optional
$ printf 'gates\nnot\nused\n' | sed '/e/{s/s/*/g; s/t/*/g}'
ga*e*
not
u*ed
```

Command grouping is an easy way to construct conditional AND of multiple search strings.

```
$ # same as: grep 'in' programming_quotes.txt | grep 'not'
$ sed -n '/in/{/not/p}' programming_quotes.txt
by definition, not smart enough to debug it by Brian W. Kernighan
A language that does not affect the way you think about programming,
is not worth knowing by Alan Perlis

$ # same as: grep 'in' programming_quotes.txt | grep 'not' | grep 'you'
$ sed -n '/in/{/not/{/you/p}}' programming_quotes.txt
A language that does not affect the way you think about programming,

$ # same as: grep 'not' programming_quotes.txt | grep -v 'you'
$ sed -n '/not/{/you/!p}' programming_quotes.txt
by definition, not smart enough to debug it by Brian W. Kernighan
is not worth knowing by Alan Perlis
```

Other solutions using alternation feature of regular expressions and `sed` 's control structures will be discussed later.

Line addressing

Line numbers can also be used as filtering criteria.

```
$ # here, 3 represents the address for print command
$ # same as: head -n3 programming_quotes.txt | tail -n1 and sed '3!d'
$ sed -n '3p' programming_quotes.txt
by definition, not smart enough to debug it by Brian W. Kernighan
```

```
$ # print 2nd and 5th line
$ sed -n '2p; 5p' programming_quotes.txt
Therefore, if you write the code as cleverly as possible, you are,
Some people, when confronted with a problem, think - I know, I will

$ # substitution only on 2nd line
$ printf 'gates\nnot\nused\n' | sed '2 s/t/*/g'
gates
no*
used
```

As a special case, `$` indicates the last line of the input.

```
$ # same as: tail -n1 programming_quotes.txt
$ sed -n '$p' programming_quotes.txt
naming things, and off-by-1 errors by Leon Bambrick
```

For large input files, use `q` command to avoid processing unnecessary input lines.

```
$ seq 3542 4623452 | sed -n '2452{p; q}'
5993
$ seq 3542 4623452 | sed -n '250p; 2452{p; q}'
3791
5993

$ # here is a sample time comparison
$ time seq 3542 4623452 | sed -n '2452{p; q}' > f1
real    0m0.003s
$ time seq 3542 4623452 | sed -n '2452p' > f2
real    0m0.256s
```

Mimicking `head` command using line addressing and `q` command.

```
$ # same as: seq 23 45 | head -n5
$ seq 23 45 | sed '5q'
23
24
25
26
27
```

Print only line number

The `=` command will display the line numbers of matching lines.

```
$ # gives both line number and matching line
$ grep -n 'not' programming_quotes.txt
3:by definition, not smart enough to debug it by Brian W. Kernighan
8:A language that does not affect the way you think about programming,
9:is not worth knowing by Alan Perlis
```

```
$ # gives only line number of matching line
$ # note the use of -n option to avoid default printing
$ sed -n '/not/=' programming_quotes.txt
3
8
9
```

If needed, matching line can also be printed. But there will be a newline character between the matching line and line number.

```
$ sed -n '/off/{=; p}' programming_quotes.txt
12
naming things, and off-by-1 errors by Leon Bambrick

$ sed -n '/off/{p; =}' programming_quotes.txt
naming things, and off-by-1 errors by Leon Bambrick
12
```

Address range

So far, filtering has been based on specific line number or lines matching the given `/REGEXP/FLAGS` pattern. Address range gives the ability to define a starting address and an ending address, separated by a comma.

```
$ # note that all the matching ranges are printed
$ sed -n '/are/,/by/p' programming_quotes.txt
Therefore, if you write the code as cleverly as possible, you are,
by definition, not smart enough to debug it by Brian W. Kernighan
There are 2 hard problems in computer science: cache invalidation,
naming things, and off-by-1 errors by Leon Bambrick

$ # same as: sed -n '3,8!p'
$ seq 15 24 | sed '3,8d'
15
16
23
24
```

Line numbers and string matching can be mixed.

```
$ sed -n '5,/use/p' programming_quotes.txt
Some people, when confronted with a problem, think - I know, I will
use regular expressions. Now they have two problems by Jamie Zawinski

$ # same as: sed '/smart/Q'
$ # inefficient, but this will work for multiple file inputs
$ sed '/smart/,$d' programming_quotes.txt
Debugging is twice as hard as writing the code in the first place.
Therefore, if you write the code as cleverly as possible, you are,
```

If the second filter condition doesn't match, lines starting from first condition to last line of

the input will be matched.

```
$ # there's a line containing 'affect' but doesn't have matching pair
$ sed -n '/affect/,/XYZ/p' programming_quotes.txt
A language that does not affect the way you think about programming,
is not worth knowing by Alan Perlis

There are 2 hard problems in computer science: cache invalidation,
naming things, and off-by-1 errors by Leon Bambrick
```

The second address will always be used as a filtering condition only from the line that comes after the line that satisfied the first address. For example, if the same search pattern is used for both the addresses, there'll be at least two lines in output (provided there are lines in the input after the first matching line).

```
$ # there's no line containing 'worth' after the 9th line
$ # so, rest of the file gets matched
$ sed -n '9,/worth/p' programming_quotes.txt
is not worth knowing by Alan Perlis

There are 2 hard problems in computer science: cache invalidation,
naming things, and off-by-1 errors by Leon Bambrick
```

As a special case, the first address can be `0` if the second one is a search pattern. This allows the search pattern to be matched against first line of the file.

```
$ # same as: sed '/in/q'
$ # inefficient, but this will work for multiple file inputs
$ sed -n '0,/in/p' programming_quotes.txt
Debugging is twice as hard as writing the code in the first place.

$ # same as: sed '/not/q'
$ sed -n '0,/not/p' programming_quotes.txt
Debugging is twice as hard as writing the code in the first place.
Therefore, if you write the code as cleverly as possible, you are,
by definition, not smart enough to debug it by Brian W. Kernighan
```

Relative addressing

Prefixing `+` to line number as the second address gives relative filtering. This is similar to using `grep -A<num> --no-group-separator` but `grep` will start a new group if a line matches within context lines.

```
$ # line matching 'not' and 2 lines after
$ # won't be same as: grep -A2 --no-group-separator 'not'
$ sed -n '/not/,+2p' programming_quotes.txt
by definition, not smart enough to debug it by Brian W. Kernighan

Some people, when confronted with a problem, think - I know, I will
A language that does not affect the way you think about programming,
is not worth knowing by Alan Perlis
```

```
$ # the first address can be a line number too
$ # helpful when it is programmatically constructed in a script
$ sed -n '5,+1p' programming_quotes.txt
Some people, when confronted with a problem, think - I know, I will
use regular expressions. Now they have two problems by Jamie Zawinski
```

You can construct an arithmetic progression with start and step values separated by the `~` symbol. `i~j` will filter lines numbered `i+0j`, `i+1j`, `i+2j`, `i+3j`, etc. So, `1~2` means all odd numbered lines and `5~3` means 5th, 8th, 11th, etc.

```
$ # print even numbered lines
$ seq 10 | sed -n '2~2p'
2
4
6
8
10

$ # delete lines numbered 2+0*4, 2+1*4, 2+2*4, etc
$ seq 7 | sed '2~4d'
1
3
4
5
7
```

If `i,~j` is used (note the `,`) then the meaning changes completely. After the start address, the closest line number which is a multiple of `j` will mark the end address. The start address can be specified using search pattern as well.

```
$ # here, closest multiple of 4 is 4th line
$ seq 10 | sed -n '2,~4p'
2
3
4
$ # here, closest multiple of 4 is 8th line
$ seq 10 | sed -n '5,~4p'
5
6
7
8

$ # line matching on 'regular' is 6th line, so ending is 9th line
$ sed -n '/regular/,~3p' programming_quotes.txt
use regular expressions. Now they have two problems by Jamie Zawinski

A language that does not affect the way you think about programming,
is not worth knowing by Alan Perlis
```

n and N commands

So far, the commands used have all been processing only one line at a time. The address range option provides the ability to act upon a group of lines, but the commands still operate one line at a time for that group. There are cases when you want a command to handle a string that contains multiple lines. As mentioned in the preface, this book will not cover advanced commands related to multiline processing and I highly recommend using `awk` or `perl` for such scenarios. However, this section will introduce two commands `n` and `N` which are relatively easier to use and will be seen in coming chapters as well.

This is also a good place to give more details about how `sed` works. Quoting from sed manual: How sed Works:

> sed maintains two data buffers: the active pattern space, and the auxiliary hold space. Both are initially empty.
> sed operates by performing the following cycle on each line of input: first, sed reads one line from the input stream, removes any trailing newline, and places it in the pattern space. Then commands are executed; each command can have an address associated to it: addresses are a kind of condition code, and a command is only executed if the condition is verified before the command is to be executed.
> When the end of the script is reached, unless the -n option is in use, the contents of pattern space are printed out to the output stream, adding back the trailing newline if it was removed. Then the next cycle starts for the next input line.

The **pattern space** buffer has only contained single line of input in all the examples seen so far. By using `n` and `N` commands, you can change the contents of pattern space and use commands to act upon entire contents of this data buffer. For example, you can perform substitution on two or more lines at once.

First up, the `n` command. Quoting from sed manual: Often-Used Commands:

> If auto-print is not disabled, print the pattern space, then, regardless, replace the pattern space with the next line of input. If there is no more input then sed exits without processing any more commands.

```
$ # same as: sed -n '2~2p'
$ # n will replace pattern space with next line of input
$ # as -n option is used, the replaced line won't be printed
$ # then the new line is printed as p command is used
$ seq 10 | sed -n 'n; p'
2
4
6
8
10

$ # if line contains 't', replace pattern space with next line
$ # substitute all 't' with 'TTT' for the new line thus fetched
$ # note that 't' wasn't substituted in the line that got replaced
$ # replaced pattern space gets printed as -n option is NOT used here
```

```
$ printf 'gates\nnot\nused\n' | sed '/t/{n; s/t/TTT/g}'
gates
noTTT
used
```

Next, the `N` command. Quoting from sed manual: Less Frequently-Used Commands:

> Add a newline to the pattern space, then append the next line of input to the pattern space. If there is no more input then sed exits without processing any more commands. When -z is used, a zero byte (the ascii 'NUL' character) is added between the lines (instead of a new line).

```
$ # append next line to pattern space
$ # and then replace newline character with colon character
$ seq 7 | sed 'N; s/\n/:/'
1:2
3:4
5:6
7

$ # if line contains 'at', the next line gets appended to pattern space
$ # then the substitution is performed on the two lines in the buffer
$ printf 'gates\nnot\nused\n' | sed '/at/{N; s/s\nnot/d/}'
gated
used
```

See also sed manual: N command on the last line. Escape sequences like `\n` will be discussed in detail later.

See grymoire: sed tutorial if you wish to explore about the data buffers in detail and learn about the various multiline commands.

Cheatsheet and summary

Note	Description
`ADDR cmd`	Execute cmd only if input line satisfies the ADDR condition `ADDR` can be REGEXP or line number or a combination of them
`/at/d`	delete all lines based on the given REGEXP
`/at/!d`	don't delete lines matching the given REGEXP
`/twice/p`	print all lines based on the given REGEXP as print is default action, usually `p` is paired with `-n` option
`/not/ s/in/**/gp`	substitute only if line matches given REGEXP and print only if substitution succeeds
`/if/q`	quit immediately after printing current pattern space further input files, if any, won't be processed

Note	Description
`/if/Q`	quit immediately without printing current pattern space
`/at/q2`	both `q` and `Q` can additionally use `0-255` as exit code
`-e 'cmd1' -e 'cmd2'`	execute multiple commands one after the other
`cmd1; cmd2`	execute multiple commands one after the other
	note that not all commands can be constructed this way
	commands can also be separated by literal newline character
`ADDR {cmds}`	group one or more commands to be executed for given ADDR
	groups can be nested as well
	ex: `/in/{/not/{/you/p}}` conditional AND of 3 REGEXPs
`2p`	line addressing, print only 2nd line
`$`	special address to indicate last line of input
`2452{p; q}`	quit early to avoid processing unnecessary lines
`/not/=`	print line number instead of matching line
`ADDR1,ADDR2`	start and end addresses to operate upon
	if ADDR2 doesn't match, lines till end of file gets processed
`/are/,/by/p`	print all groups of line matching the REGEXPs
`3,8d`	delete lines numbered 3 to 8
`5,/use/p`	line number and REGEXP can be mixed
`0,/not/p`	inefficient equivalent of `/not/q` but works for multiple files
`ADDR,+N`	all lines matching the ADDR and `N` lines after
`i~j`	arithmetic progression with `i` as start and `j` as step
`ADDR,~j`	closest multiple of `j` wrt line matching the ADDR
pattern space	active data buffer, commands work on this content
`n`	if `-n` option isn't used, pattern space gets printed
	and then pattern space is replaced with the next line of input
	exit without executing other commands if there's no more input
`N`	add newline (or NUL for `-z`) to the pattern space
	and then append next line of input
	exit without executing other commands if there's no more input

This chapter introduced the filtering capabilities of `sed` and how it can be combined with `sed` commands to process only lines of interest instead of entire input file. Filtering can be specified using a REGEXP, line number or a combination of them. You also learnt various ways to compose multiple `sed` commands. In the next chapter, you will learn syntax and features of regular expressions as implemented in `sed` command.

Exercises

a) Remove only the third line of given input.

```
$ seq 34 37 | sed ##### add your solution here
34
35
37
```

b) Display only fourth, fifth, sixth and seventh lines for the given input.

```
$ seq 65 78 | sed ##### add your solution here
68
69
70
71
```

c) For the input file `addr.txt`, replace all occurrences of `are` with `are not` and `is` with `is not` only from line number **4** till end of file. Also, only the lines that were changed should be displayed in the output.

```
$ cat addr.txt
Hello World
How are you
This game is good
Today is sunny
12345
You are funny

$ sed ##### add your solution here
Today is not sunny
You are not funny
```

d) Use `sed` to get the output shown below for the given input. You'll have to first understand the logic behind input to output transformation and then use commands introduced in this chapter to construct a solution.

```
$ seq 15 | sed ##### add your solution here
2
4
7
9
12
14
```

e) For the input file `addr.txt`, display all lines from start of the file till the first occurrence of `game`.

```
$ sed ##### add your solution here
Hello World
How are you
This game is good
```

f) For the input file `addr.txt`, display all lines that contain `is` but not `good`.

```
$ sed ##### add your solution here
Today is sunny
```

g) See Gotchas and Tricks chapter and correct the command to get the output as shown below.

```
$ # wrong output
$ seq 11 | sed 'N; N; s/\n/-/g'
1-2-3
4-5-6
```

```
7-8-9
10
11

$ # expected output
$ seq 11 | sed ##### add your solution here
1-2-3
4-5-6
7-8-9
10-11
```

h) For the input file `addr.txt` , add line numbers in the format as shown below.

```
$ sed ##### add your solution here
1
Hello World
2
How are you
3
This game is good
4
Today is sunny
5
12345
6
You are funny
```

i) For the input file `addr.txt` , print all lines that contain `are` and the line that comes after such a line, if any.

```
$ sed ##### add your solution here
How are you
This game is good
You are funny
```

Bonus: For the above input file, will `sed -n '/is/,+1 p' addr.txt` produce identical results as `grep -A1 'is' addr.txt` ? If not, why?

j) Print all lines if their line numbers follow the sequence `1, 15, 29, 43, etc` but not if the line contains `4` in it.

```
$ seq 32 100 | sed ##### add your solution here
32
60
88
```

BRE/ERE Regular Expressions

This chapter will cover Basic and Extended Regular Expressions as implemented in `GNU sed`. Though not strictly conforming to POSIX specifications, most of it is applicable to other `sed` implementations as well. Unless otherwise indicated, examples and descriptions will assume ASCII input.

By default, `sed` treats the search pattern as Basic Regular Expression (BRE). Using `-E` option will enable Extended Regular Expression (ERE). Older versions used `-r` for ERE, which can still be used, but `-E` is more portable. In `GNU sed`, BRE and ERE only differ in how metacharacters are applied, there's no difference in features.

Line Anchors

Instead of matching anywhere in the line, restrictions can be specified. These restrictions are made possible by assigning special meaning to certain characters and escape sequences. The characters with special meaning are known as **metacharacters** in regular expressions parlance. In case you need to match those characters literally, you need to escape them with a `\` (discussed in Matching the metacharacters section).

There are two line anchors:

- `^` metacharacter restricts the matching to start of line
- `$` metacharacter restricts the matching to end of line

```
$ # lines starting with 'sp'
$ printf 'spared no one\npar\nspar\n' | sed -n '/^sp/p'
spared no one
spar

$ # lines ending with 'ar'
$ printf 'spared no one\npar\nspar\n' | sed -n '/ar$/p'
par
spar

$ # change only whole line 'par'
$ printf 'spared no one\npar\nspar\n' | sed 's/^par$/PAR/'
spared no one
PAR
spar
```

The anchors can be used by themselves as a pattern. Helps to insert text at start or end of line, emulating string concatenation operations. These might not feel like useful capability, but combined with other features they become quite a handy tool.

```
$ printf 'spared no one\npar\nspar\n' | sed 's/^/* /'
* spared no one
* par
* spar

$ # append only if line doesn't contain space characters
```

```
$ printf 'spared no one\npar\nspar\n' | sed '/ /! s/$/./'
spared no one
par.
spar.
```

Word Anchors

The second type of restriction is word anchors. A word character is any alphabet (irrespective of case), digit and the underscore character. You might wonder why there are digits and underscores as well, why not only alphabets? This comes from variable and function naming conventions — typically alphabets, digits and underscores are allowed. So, the definition is more programming oriented than natural language.

The escape sequence `\b` denotes a word boundary. This works for both start of word and end of word anchoring. Start of word means either the character prior to the word is a non-word character or there is no character (start of line). Similarly, end of word means the character after the word is a non-word character or no character (end of line). This implies that you cannot have word boundary without a word character.

As an alternate, you can use `\<` to indicate start of word anchor and `\>` to indicate end of word anchor. Using `\b` is preferred as it is more commonly used in other regular expression implementations and has `\B` as its opposite.

`\bREGEXP\b` behaves a bit differently than `\<REGEXP\>` . See Gotchas and Tricks chapter for details.

```
$ cat word_anchors.txt
sub par
spar
apparent effort
two spare computers
cart part tart mart

$ # words starting with 'par'
$ sed -n '/\bpar/p' word_anchors.txt
sub par
cart part tart mart

$ # words ending with 'par'
$ sed -n '/par\b/p' word_anchors.txt
sub par
spar

$ # only whole word 'par'
$ sed -n 's/\bpar\b/***/p' word_anchors.txt
sub ***
```

The word boundary has an opposite anchor too. `\B` matches wherever `\b` doesn't match. This duality will be seen with some other escape sequences too.

> ⚠ Negative logic is handy in many text processing situations. But use it with care, you might end up matching things you didn't intend.

```
$ # match 'par' if it is surrounded by word characters
$ sed -n '/\Bpar\B/p' word_anchors.txt
apparent effort
two spare computers

$ # match 'par' but not as start of word
$ sed -n '/\Bpar/p' word_anchors.txt
spar
apparent effort
two spare computers

$ # match 'par' but not as end of word
$ sed -n '/par\B/p' word_anchors.txt
apparent effort
two spare computers
cart part tart mart

$ echo 'copper' | sed 's/\b/:/g'
:copper:
$ echo 'copper' | sed 's/\B/:/g'
c:o:p:p:e:r
```

Alternation

Many a times, you'd want to search for multiple terms. In a conditional expression, you can use the logical operators to combine multiple conditions. With regular expressions, the `|` metacharacter is similar to logical OR. The regular expression will match if any of the expression separated by `|` is satisfied. These can have their own independent anchors as well.

Alternation is similar to using multiple `-e` option, but provides more flexibility with regular expression features. The `|` metacharacter syntax varies between BRE and ERE. Quoting from the manual:

> In GNU sed, the only difference between basic and extended regular expressions is in the behavior of a few special characters: `?` , `+` , parentheses, braces (`{}`), and `|` .

```
$ # BRE vs ERE
$ sed -n '/two\|sub/p' word_anchors.txt
sub par
two spare computers
$ sed -nE '/two|sub/p' word_anchors.txt
```

```
sub par
two spare computers

$ # either 'cat' or 'dog' or 'fox'
$ # note the use of 'g' flag for multiple replacements
$ echo 'cats dog bee parrot foxed' | sed -E 's/cat|dog|fox/--/g'
--s -- bee parrot --ed

$ # lines with whole word 'par' or lines ending with 's'
$ sed -nE '/\bpar\b|s$/p' word_anchors.txt
sub par
two spare computers
```

There's some tricky situations when using alternation. If it is used for filtering a line, there is no ambiguity. However, for use cases like substitution, it depends on a few factors. Say, you want to replace `are` or `spared` — which one should get precedence? The bigger word `spared` or the substring `are` inside it or based on something else?

The alternative which matches earliest in the input gets precedence.

```
$ # here, the output will be same irrespective of alternation order
$ # note that 'g' flag isn't used here, so only first match gets replaced
$ echo 'cats dog bee parrot foxed' | sed -E 's/bee|parrot|at/--/'
c--s dog bee parrot foxed
$ echo 'cats dog bee parrot foxed' | sed -E 's/parrot|at|bee/--/'
c--s dog bee parrot foxed
```

In case of matches starting from same location, for example `spar` and `spared`, the longest matching portion gets precedence. Unlike other regular expression implementations, left-to-right priority for alternation comes into play only if length of the matches are the same. See Longest match wins and Backreferences sections for more examples. See regular-expressions: alternation for more information on this topic.

```
$ echo 'spared party parent' | sed -E 's/spa|spared/**/g'
** party parent
$ echo 'spared party parent' | sed -E 's/spared|spa/**/g'
** party parent

$ # other implementations like 'perl' have left-to-right priority
$ echo 'spared party parent' | perl -pe 's/spa|spared/**/'
**red party parent
```

Grouping

Often, there are some common things among the regular expression alternatives. It could be common characters or qualifiers like the anchors. In such cases, you can group them using a pair of parentheses metacharacters. Similar to `a(b+c)d = abd+acd` in maths, you get `a(b|c)d = abd|acd` in regular expressions.

```
$ # without grouping
$ printf 'red\nreform\nread\narrest\n' | sed -nE '/reform|rest/p'
reform
arrest
$ # with grouping
$ printf 'red\nreform\nread\narrest\n' | sed -nE '/re(form|st)/p'
reform
arrest

$ # without grouping
$ printf 'sub par\nspare\npart time\n' | sed -nE '/\bpar\b|\bpart\b/p'
sub par
part time
$ # taking out common anchors
$ printf 'sub par\nspare\npart time\n' | sed -nE '/\b(par|part)\b/p'
sub par
part time
$ # taking out common characters as well
$ # you'll later learn a better technique instead of using empty alternate
$ printf 'sub par\nspare\npart time\n' | sed -nE '/\bpar(|t)\b/p'
sub par
part time
```

Matching the metacharacters

You have seen a few metacharacters and escape sequences that help to compose a regular expression. To match the metacharacters literally, i.e. to remove their special meaning, prefix those characters with a `\` character. To indicate a literal `\` character, use `\\` . Some of the metacharacters, like the line anchors, lose their special meaning when not used in their customary positions. If there are many metacharacters to be escaped, try to work out if the command can be simplified by switching between ERE and BRE.

```
$ # line anchors aren't special away from customary positions
$ echo 'a^2 + b^2 - C*3' | sed -n '/b^2/p'
a^2 + b^2 - C*3
$ echo '$a = $b + $c' | sed -n '/$b/p'
$a = $b + $c
$ # escape line anchors to match them literally at customary positions
$ echo '$a = $b + $c' | sed 's/\$//g'
a = b + c

$ # BRE vs ERE
$ printf '(a/b) + c\n3 + (a/b) - c\n' | sed -n '/^(a\/b)/p'
(a/b) + c
$ printf '(a/b) + c\n3 + (a/b) - c\n' | sed -nE '/^\(a\/b\)/p'
(a/b) + c
```

Handling the metacharacters in replacement section will be discussed in Backreferences section.

Using different delimiters

The `/` character is idiomatically used as the delimiter for REGEXP. But any character other than `\` and the newline character can be used instead. This helps to avoid or reduce the need for escaping delimiter characters. The syntax is simple for substitution and transliteration commands, just use a different character instead of `/` .

```
$ # instead of this
$ echo '/home/learnbyexample/reports' | sed 's/\/home\/learnbyexample\//~\//'
~/reports
$ # use a different delimiter
$ echo '/home/learnbyexample/reports' | sed 's#/home/learnbyexample/#~/#'
~/reports

$ echo 'a/b/c/d' | sed 'y/a\/d/1-4/'
1-b-c-4
$ echo 'a/b/c/d' | sed 'y,a/d,1-4,'
1-b-c-4
```

For address matching, syntax is a bit different, the first delimiter has to be escaped. For address ranges, start and end REGEXP can have different delimiters, as they are independent.

```
$ printf '/foo/bar/1\n/foo/baz/1\n'
/foo/bar/1
/foo/baz/1

$ # here ; is used as the delimiter
$ printf '/foo/bar/1\n/foo/baz/1\n' | sed -n '\;/foo/bar/;p'
/foo/bar/1
```

See also a bit of history on why / is commonly used as delimiter.

The dot meta character

The dot metacharacter serves as a placeholder to match any character (including newline character). Later you'll learn how to define your own custom placeholder for limited set of characters.

```
$ # 3 character sequence starting with 'c' and ending with 't'
$ echo 'tac tin cot abc:tyz excited' | sed 's/c.t/-/g'
ta-in - ab-yz ex-ed

$ # any character followed by 3 and again any character
$ printf '42\t35\n' | sed 's/.3.//'
42

$ # N command is handy here to show that . matches \n as well
$ printf 'abc\nxyz\n' | sed 'N; s/c.x/ /'
ab yz
```

Quantifiers

As an analogy, alternation provides logical OR. Combining the dot metacharacter `.` and quantifiers (and alternation if needed) paves a way to perform logical AND. For example, to check if a string matches two patterns with any number of characters in between. Quantifiers can be applied to both characters and groupings. Apart from ability to specify exact quantity and bounded range, these can also match unbounded varying quantities.

First up, the `?` metacharacter which quantifies a character or group to match `0` or `1` times. This helps to define optional patterns and build terser patterns compared to groupings for some cases.

```
$ # same as: sed -E 's/\b(fe.d|fed)\b/X/g'
$ # BRE version: sed 's/fe.\?d\b/X/g'
$ echo 'fed fold fe:d feeder' | sed -E 's/\bfe.?d\b/X/g'
X fold X feeder

$ # same as: sed -nE '/\bpar(|t)\b/p'
$ printf 'sub par\nspare\npart time\n' | sed -nE '/\bpart?\b/p'
sub par
part time

$ # same as: sed -E 's/part|parrot/X/g'
$ echo 'par part parrot parent' | sed -E 's/par(ro)?t/X/g'
par X X parent
$ # same as: sed -E 's/part|parrot|parent/X/g'
$ echo 'par part parrot parent' | sed -E 's/par(en|ro)?t/X/g'
par X X X

$ # both '<' and '\<' are replaced with '\<'
$ echo 'blah \< foo bar < blah baz <' | sed -E 's/\\?</\\</g'
blah \< foo bar \< blah baz \<
```

The `*` metacharacter quantifies a character or group to match `0` or more times. There is no upper bound, more details will be discussed in the next section.

```
$ # 'f' followed by zero or more of 'e' followed by 'd'
$ echo 'fd fed fod fe:d feeeeder' | sed 's/fe*d/X/g'
X X fod fe:d Xer

$ # zero or more of '1' followed by '2'
$ echo '3111111111125111142' | sed 's/1*2/-/g'
3-511114-
```

The `+` metacharacter quantifies a character or group to match `1` or more times. Similar to `*` quantifier, there is no upper bound.

```
$ # 'f' followed by one or more of 'e' followed by 'd'
$ # BRE version: sed 's/fe\+d/X/g'
$ echo 'fd fed fod fe:d feeeeder' | sed -E 's/fe+d/X/g'
fd X fod fe:d Xer
```

```
$ # 'f' followed by at least one of 'e' or 'o' or ':' followed by 'd'
$ echo 'fd fed fod fe:d feeeeder' | sed -E 's/f(e|o|:)+d/X/g'
fd X X X Xer

$ # one or more of '1' followed by optional '4' and then '2'
$ echo '3111111111125111142' | sed -E 's/1+4?2/-/g'
3-5-
```

You can specify a range of integer numbers, both bounded and unbounded, using `{}` metacharacters. There are four ways to use this quantifier as listed below:

Pattern	Description
`{m,n}`	match `m` to `n` times
`{m,}`	match at least `m` times
`{,n}`	match up to `n` times (including `0` times)
`{n}`	match exactly `n` times

```
$ # note that inside {} space is not allowed around ,
$ # BRE version: sed 's/ab\{1,4\}c/X/g'
$ echo 'ac abc abbc abbbc abbbbbbbbc' | sed -E 's/ab{1,4}c/X/g'
ac X X X abbbbbbbbc

$ echo 'ac abc abbc abbbc abbbbbbbbc' | sed -E 's/ab{3,}c/X/g'
ac abc abbc X X

$ echo 'ac abc abbc abbbc abbbbbbbbc' | sed -E 's/ab{,2}c/X/g'
X X X abbbc abbbbbbbbc

$ echo 'ac abc abbc abbbc abbbbbbbbc' | sed -E 's/ab{3}c/X/g'
ac abc abbc X abbbbbbbbc
```

The `{}` metacharacters have to be escaped to match them literally. However, unlike `()` metacharacters, escaping `{` alone is enough.

Next up, how to construct conditional AND using dot metacharacter and quantifiers. To allow matching in any order, you'll have to bring in alternation as well. But, for more than 3 patterns, the combinations become too many to write and maintain.

```
$ # match 'Error' followed by zero or more characters followed by 'valid'
$ echo 'Error: not a valid input' | sed -n '/Error.*valid/p'
Error: not a valid input

$ # 'cat' followed by 'dog' or 'dog' followed by 'cat'
$ echo 'two cats and a dog' | sed -E 's/cat.*dog|dog.*cat/pets/'
two pets
$ echo 'two dogs and a cat' | sed -E 's/cat.*dog|dog.*cat/pets/'
two pets
```

Longest match wins

You've already seen an example with alternation, where the longest matching portion was chosen if two alternatives started from same location. For example `spar|spared` will result in `spared` being chosen over `spar`. The same applies whenever there are two or more matching possibilities with quantifiers starting from same location. For example, `f.?o` will match `foo` instead of `fo` if the input string to match is `foot`.

```
$ # longest match among 'foo' and 'fo' wins here
$ echo 'foot' | sed -E 's/f.?o/X/'
Xt
$ # everything will match here
$ echo 'car bat cod map scat dot abacus' | sed 's/.*/X/'
X

$ # longest match happens when (1|2|3)+ matches up to '1233' only
$ # so that '12baz' can match as well
$ echo 'foo123312baz' | sed -E 's/o(1|2|3)+(12baz)?/X/'
foX
$ # in other implementations like 'perl', that is not the case
$ # quantifiers match as much as possible, but precedence is left to right
$ echo 'foo123312baz' | perl -pe 's/o(1|2|3)+(12baz)?/X/'
foXbaz
```

While determining the longest match, overall regular expression matching is also considered. That's how `Error.*valid` example worked. If `.*` had consumed everything after `Error`, there wouldn't be any more characters to try to match `valid`. So, among the varying quantity of characters to match for `.*`, the longest portion that satisfies the overall regular expression is chosen. Something like `a.*b` will match from first `a` in the input string to the last `b` in the string. In other implementations, like `perl`, this is achieved through a process called **backtracking**. Both approaches have their own advantages and disadvantages and have cases where the pattern can result in exponential time consumption.

```
$ # from start of line to last 'm' in the line
$ echo 'car bat cod map scat dot abacus' | sed 's/.*m/-/'
-ap scat dot abacus

$ # from first 'b' to last 't' in the line
$ echo 'car bat cod map scat dot abacus' | sed 's/b.*t/-/'
car - abacus

$ # from first 'b' to last 'at' in the line
$ echo 'car bat cod map scat dot abacus' | sed 's/b.*at/-/'
car - dot abacus

$ # here 'm*' will match 'm' zero times as that gives the longest match
$ echo 'car bat cod map scat dot abacus' | sed 's/a.*m*/-/'
c-
```

Character classes

To create a custom placeholder for limited set of characters, enclose them inside `[]` metacharacters. It is similar to using single character alternations inside a grouping, but with added flexibility and features. Character classes have their own versions of metacharacters and provide special predefined sets for common use cases. Quantifiers are also applicable to character classes.

```
$ # same as: sed -nE '/cot|cut/p' and sed -nE '/c(o|u)t/p'
$ printf 'cute\ncat\ncot\ncoat\ncost\nscuttle\n' | sed -n '/c[ou]t/p'
cute
cot
scuttle

$ # same as: sed -nE '/.(a|e|o)+t/p'
$ printf 'meeting\ncute\nboat\nat\nfoot\n' | sed -nE '/.[aeo]+t/p'
meeting
boat
foot

$ # same as: sed -E 's/\b(s|o|t)(o|n)\b/X/g'
$ echo 'no so in to do on' | sed -E 's/\b[sot][on]\b/X/g'
no X in X do X

$ # lines made up of letters 'o' and 'n', line length at least 2
$ # words.txt contains dictionary words, one word per line
$ sed -nE '/^[on]{2,}$/p' words.txt
no
non
noon
on
```

Character classes have their own metacharacters to help define the sets succinctly. Metacharacters outside of character classes like `^` , `$` , `()` etc either don't have special meaning or have completely different one inside the character classes. First up, the `-` metacharacter that helps to define a range of characters instead of having to specify them all individually.

```
$ # same as: sed -E 's/[0123456789]+/-/g'
$ echo 'Sample123string42with777numbers' | sed -E 's/[0-9]+/-/g'
Sample-string-with-numbers

$ # whole words made up of lowercase alphabets and digits only
$ echo 'coat Bin food tar12 best' | sed -E 's/\b[a-z0-9]+\b/X/g'
X Bin X X X

$ # whole words made up of lowercase alphabets, starting with 'p' to 'z'
$ echo 'road i post grip read eat pit' | sed -E 's/\b[p-z][a-z]*\b/X/g'
X i X grip X eat X
```

Character classes can also be used to construct numeric ranges. However, it is easy to miss corner cases and some ranges are complicated to design. See also regular-expressions: Match-

ing Numeric Ranges with a Regular Expression.

```
$ # numbers between 10 to 29
$ echo '23 154 12 26 34' | sed -E 's/\b[12][0-9]\b/X/g'
X 154 X X 34
$ # numbers >= 100 with optional leading zeros
$ echo '0501 035 154 12 26 98234' | sed -E 's/\b0*[1-9][0-9]{2,}\b/X/g'
X 035 X 12 26 X
```

Next metacharacter is `^` which has to specified as the first character of the character class. It negates the set of characters, so all characters other than those specified will be matched. As highlighted earlier, handle negative logic with care, you might end up matching more than you wanted.

```
$ # replace all non-digits
$ echo 'Sample123string42with777numbers' | sed -E 's/[^0-9]+/-/g'
-123-42-777-

$ # delete last two columns based on a delimiter
$ echo 'foo:123:bar:baz' | sed -E 's/(:[^:]+){2}$//'
foo:123

$ # sequence of characters surrounded by unique character
$ echo 'I like "mango" and "guava"' | sed -E 's/"[^"]+"/X/g'
I like X and X

$ # sometimes it is simpler to positively define a set than negation
$ # same as: sed -nE '/^[^aeiou]*$/p'
$ printf 'tryst\nfun\nglyph\npity\nwhy\n' | sed '/[aeiou]/d'
tryst
glyph
why
```

Some commonly used character sets have predefined escape sequences:

- `\w` matches all **word** characters `[a-zA-Z0-9_]` (recall the description for word boundaries)
- `\W` matches all non-word characters (recall duality seen earlier, like `\b` and `\B`)
- `\s` matches all **whitespace** characters: tab, newline, vertical tab, form feed, carriage return and space
- `\S` matches all non-whitespace characters

These escape sequences cannot be used inside character classes. Also, as mentioned earlier, these definitions assume ASCII input.

`sed` doesn't support `\d` and `\D`, commonly featured in other implementations as a shortcut for all the digits and non-digits.

```
$ # match all non-word characters
$ echo 'load;err_msg--\nant,r2..not' | sed -E 's/\W+/-/g'
load-err_msg-nant-r2-not
```

```
$ # replace all sequences of whitespaces with single space
$ printf 'hi  \v\f  there.\thave   \ra nice\t\tday\n' | sed -E 's/\s+/ /g'
hi there. have a nice day

$ # \w would simply match \ and w inside character classes
$ echo 'w=y\x+9*3' | sed 's/[\w=]//g'
yx+9*3
```

A **named character set** is defined by a name enclosed between `[:` and `:]` and has to be used within a character class `[]` , along with any other characters as needed.

Named set	Description
`[:digit:]`	`[0-9]`
`[:lower:]`	`[a-z]`
`[:upper:]`	`[A-Z]`
`[:alpha:]`	`[a-zA-Z]`
`[:alnum:]`	`[0-9a-zA-Z]`
`[:xdigit:]`	`[0-9a-fA-F]`
`[:cntrl:]`	control characters — first 32 ASCII characters and 127th (DEL)
`[:punct:]`	all the punctuation characters
`[:graph:]`	`[:alnum:]` and `[:punct:]`
`[:print:]`	`[:alnum:]` , `[:punct:]` and space
`[:blank:]`	space and tab characters
`[:space:]`	whitespace characters, same as `\s`

```
$ echo 'err_msg xerox ant m_2 P2 load1 eel' | sed -E 's/\b[[:lower:]]+\b/X/g'
err_msg X X m_2 P2 load1 X

$ echo 'err_msg xerox ant m_2 P2 load1 eel' | sed -E 's/\b[[:lower:]_]+\b/X/g'
X X X m_2 P2 load1 X

$ echo 'err_msg xerox ant m_2 P2 load1 eel' | sed -E 's/\b[[:alnum:]]+\b/X/g'
err_msg X X m_2 X X X

$ echo ',pie tie#ink-eat_42' | sed -E 's/[^[:punct:]]+//g'
,#-_
```

Specific placement is needed to match character class metacharacters literally.

Combinations like `[.` or `[:` cannot be used together to mean two individual characters, as they have special meaning within `[]` . See sed manual: Character Classes and Bracket Expressions for more details.

```
$ # - should be first or last character within []
$ echo 'ab-cd gh-c 12-423' | sed -E 's/[a-z-]{2,}/X/g'
X X 12-423
```

```
$ # ] should be first character within []
$ printf 'int a[5]\nfoo\n1+1=2\n' | sed -n '/[=]]/p'
$ printf 'int a[5]\nfoo\n1+1=2\n' | sed -n '/[]=]/p'
int a[5]
1+1=2

$ # to match [ use [ anywhere in the character set
$ # but not combinations like [. or [:
$ # [][] will match both [ and ]
$ echo 'int a[5]' | sed -n '/[x[.y]/p'
sed: -e expression #1, char 9: unterminated address regex
$ echo 'int a[5]' | sed -n '/[x[y.]/p'
int a[5]

$ # ^ should be other than first character within []
$ echo 'f*(a^b) - 3*(a+b)/(a-b)' | sed 's/a[+^]b/c/g'
f*(c) - 3*(c)/(a-b)
```

Escape sequences

Certain ASCII characters like tab `\t`, carriage return `\r`, newline `\n`, etc have escape sequences to represent them. Additionally, any character can be represented using their ASCII value in decimal `\dNNN` or octal `\oNNN` or hexadecimal `\xNN` formats. Unlike character set escape sequences like `\w`, these can be used inside character classes. As `\` is special inside character class, use `\\` to represent it literally (technically, this is only needed if the combination of `\` and the character(s) that follows is a valid escape sequence).

```
$ # using \t to represent tab character
$ printf 'foo\tbar\tbaz\n' | sed 's/\t/ /g'
foo bar baz
$ echo 'a b c' | sed 's/ /\t/g'
a       b       c

$ # these escape sequence work inside character class too
$ printf 'a\t\r\fb\vc\n' | sed -E 's/[\t\v\f\r]+/:/g'
a:b:c

$ # representing single quotes
$ # use \d039 and \o047 for decimal and octal respectively
$ echo "universe: '42'" | sed 's/\x27/"/g'
universe: "42"
$ echo 'universe: "42"' | sed 's/"/\x27/g'
universe: '42'
```

If a metacharacter is specified by ASCII value format in search section, it will still act as the metacharacter. However, metacharacters specified by ASCII value format in replacement section acts as a literal character. Undefined escape sequences (both search and replacement section) will be treated as the character it escapes, for example, `\e` will match `e` (not `\` and `e`).

```
$ # \x5e is ^ character, acts as line anchor here
$ printf 'cute\ncot\ncat\ncoat\n' | sed -n '/\x5eco/p'
cot
coat

$ # & metacharacter in replacement will be discussed in next section
$ # it represents entire matched portion
$ echo 'hello world' | sed 's/.*/"&"/'
"hello world"
$ # \x26 is & character, acts as literal character here
$ echo 'hello world' | sed 's/.*/"\x26"/'
"&"
```

See sed manual: Escapes for full list and details such as precedence rules.

Backreferences

The grouping metacharacters `()` are also known as **capture groups**. They are like variables, the string captured by `()` can be referred later using backreference `\N` where `N` is the capture group you want. Leftmost `(` in the regular expression is `\1`, next one is `\2` and so on up to `\9`. Backreferences can be used in both search and replacement sections. Quantifiers can be applied to backreferences as well.

```
$ # whole words that have at least one consecutive repeated character
$ # word boundaries are not needed here due to longest match wins effect
$ echo 'effort flee facade oddball rat tool' | sed -E 's/\w*(\w)\1\w*/X/g'
X X facade X rat X

$ # reduce \\ to single \ and delete if it is a single \
$ echo '\[\] and \\w and \[a-zA-Z0-9\_\]' | sed -E 's/(\\?)\\/\1/g'
[] and \w and [a-zA-Z0-9_]

$ # remove two or more duplicate words separated by space
$ # \b prevents false matches like 'the theatre', 'sand and stone' etc
$ echo 'aa a a a 42 f_1 f_1 f_13.14' | sed -E 's/\b(\w+)( \1)+\b/\1/g'
aa a 42 f_1 f_13.14

$ # 8 character lines having same 3 lowercase letters at start and end
$ sed -nE '/^([a-z]{3})..\1$/p' words.txt
mesdames
```

```
respires
restores
testates
```

As a special case, `\0` or `&` metacharacter represents entire matched string in replacement section.

```
$ # duplicate first column value as final column
$ # same as: sed -E 's/^([^,]+).*/\0,\1/'
$ echo 'one,2,3.14,42' | sed -E 's/^([^,]+).*/&,\1/'
one,2,3.14,42,one

$ # surround entire line with double quotes
$ echo 'hello world' | sed 's/.*/"&"/'
"hello world"

$ # add something at start and end of line
$ echo 'hello world' | sed 's/.*/Hi. &. Have a nice day/'
Hi. hello world. Have a nice day
```

If quantifier is applied on a pattern grouped inside `()` metacharacters, you'll need an outer `()` group to capture the matching portion. Other regular expression engines like PCRE (Perl Compatible Regular Expressions) provide non-capturing group to handle such cases. In `sed`, you'll have to work around the extra capture group.

```
$ # surround only third column with double quotes
$ # note the numbers used in replacement section
$ echo 'one,2,3.14,42' | sed -E 's/^(([^,]+,){2})([^,]+)/\1"\3"/'
one,2,"3.14",42
```

Here's an example where alternation order matters when matching portions have same length. Aim is to delete all whole words unless it starts with `g` or `p` and contains `y`. See stackoverflow: Non greedy matching in sed for another use case.

```
$ s='tryst,fun,glyph,pity,why,group'

$ # all words get deleted because \b\w+\b gets priority here
$ echo "$s" | sed -E 's/\b\w+\b|(\b[gp]\w*y\w*\b)/\1/g'
,,,,,

$ # capture group gets priority here, thus words matching the group are retained
$ echo "$s" | sed -E 's/(\b[gp]\w*y\w*\b)|\b\w+\b/\1/g'
,,glyph,pity,,
```

As `\` and `&` are special characters in replacement section, use `\\` and `\&` respectively for literal representation.

```
$ echo 'foo and bar' | sed 's/and/[&]/'
foo [and] bar
$ echo 'foo and bar' | sed 's/and/[\&]/'
foo [&] bar
```

```
$ echo 'foo and bar' | sed 's/and/\\/'
foo \ bar
```

Backreference will provide the string that was matched, not the pattern that was inside the capture group. For example, if `([0-9][a-f])` matches `3b`, then backreferencing will give `3b` and not any other valid match like `8f`, `0a` etc. This is akin to how variables behave in programming, only the result of expression stays after variable assignment, not the expression itself.

Known Bugs

Visit sed bug list for known issues.

Here's is an issue for certain usage of backreferences and quantifier that was filed by yours truly.

```
$ # takes some time and results in no output
$ # aim is to get words having two occurrences of repeated characters
$ # works if you use perl -ne 'print if /^(\w*(\w)\2\w*){2}$/'
$ sed -nE '/^(\w*(\w)\2\w*){2}$/p' words.txt | head -n5

$ # works when nesting is unrolled
$ sed -nE '/^\w*(\w)\1\w*(\w)\2\w*$/p' words.txt | head -n5
Abbott
Annabelle
Annette
Appaloosa
Appleseed
```

unix.stackexchange: Why doesn't this sed command replace the 3rd-to-last "and"? shows another interesting bug when word boundaries and group repetition are involved. Some examples are shown below. Again, workaround is to expand the group.

```
$ # wrong output
$ echo 'cocoa' | sed -nE '/(\bco){2}/p'
cocoa
$ # correct behavior, no output
$ echo 'cocoa' | sed -nE '/\bco\bco/p'

$ # wrong output, there's only 1 whole word 'it' after 'with'
$ echo 'it line with it here sit too' | sed -E 's/with(.*\bit\b){2}/XYZ/'
it line XYZ too
$ # correct behavior, input isn't modified
$ echo 'it line with it here sit too' | sed -E 's/with.*\bit\b.*\bit\b/XYZ/'
it line with it here sit too

$ # changing word boundaries to \< and \> results in a different problem
```

```
$ # this correctly doesn't modify the input
$ echo 'it line with it here sit too' | sed -E 's/with(.*\<it\>){2}/XYZ/'
it line with it here sit too
$ # this correctly modifies the input
$ echo 'it line with it here it too' | sed -E 's/with(.*\<it\>){2}/XYZ/'
it line XYZ too
$ # but this one fails to modify the input
$ echo 'it line with it here it too sit' | sed -E 's/with(.*\<it\>){2}/XYZ/'
it line with it here it too sit
```

Cheatsheet and summary

Note	Description
BRE	Basic Regular Expression, enabled by default
ERE	Extended Regular Expression, enabled using `-E` option
	note: only ERE syntax is covered below
metacharacters	characters with special meaning in REGEXP
`^`	restricts the match to start of line
`$`	restricts the match to end of line
`\b`	restricts the match to start/end of words
	word characters: alphabets, digits, underscore
`\B`	matches wherever `\b` doesn't match
`\<`	start of word anchor
`\>`	end of word anchor
`|`	combine multiple patterns as conditional OR
	each alternative can have independent anchors
	alternative which matches earliest in the input gets precedence
	and the leftmost longest portion wins in case of a tie
`()`	group pattern(s)
`a(b|c)d`	same as `abd|acd`
`\`	prefix metacharacters with `\` to match them literally
`\\`	to match `\` literally
	switching between ERE and BRE helps in some cases
`/`	idiomatically used as the delimiter for REGEXP
	any character except `\` and newline character can also be used
`.`	match any character, including the newline character
`?`	match `0` or `1` times
`*`	match `0` or more times
`+`	match `1` or more times
`{m,n}`	match `m` to `n` times
`{m,}`	match at least `m` times
`{,n}`	match up to `n` times (including `0` times)
`{n}`	match exactly `n` times
`pat1.*pat2`	any number of characters between `pat1` and `pat2`
`pat1.*pat2|pat2.*pat1`	match both `pat1` and `pat2` in any order
`[ae;o]`	match any of these characters once
	quantifiers are applicable to character classes too

Note	Description
`[3-7]`	range of characters from `3` to `7`
`[^=b2]`	match other than `=` or `b` or `2`
`[a-z-]`	`-` should be first/last character to match literally
`[+^]`	`^` shouldn't be first character
`[]=]`	`]` should be first character
	combinations like `[.` or `[:` have special meaning
`\w`	similar to `[a-zA-Z0-9_]` for matching word characters
`\s`	similar to `[ \t\n\r\f\v]` for matching whitespace characters
	`\W` and `\S` for their opposites respectively
`[:digit:]`	named character set, same as `[0-9]`
`\xNN`	represent ASCII character using hexadecimal value
	use `\dNNN` for decimal and `\oNNN` for octal
`\N`	backreference, gives matched portion of Nth capture group
	applies to both search and replacement sections
	possible values: `\1` , `\2` up to `\9`
`\0` or `&`	represents entire matched string in replacement section

Regular expressions is a feature that you'll encounter in multiple command line programs and programming languages. It is a versatile tool for text processing. Although the features provided by BRE/ERE implementation are less compared to those found in programming languages, they are sufficient for most of the tasks you'll need for command line usage. It takes a lot of time to get used to syntax and features of regular expressions, so I'll encourage you to practice a lot and maintain notes. It'd also help to consider it as a mini-programming language in itself for its flexibility and complexity. In the next chapter, you'll learn about flags that add more features to regular expressions usage.

Exercises

a) For the given input, print all lines that start with `den` or end with `ly` .

```
$ lines='lovely\n1 dentist\n2 lonely\neden\nfly away\ndent\n'
$ printf '%b' "$lines" | sed ##### add your solution here
lovely
2 lonely
dent
```

b) Replace all occurrences of `42` with `[42]` unless it is at the edge of a word. Note that **word** in these exercises have same meaning as defined in regular expressions.

```
$ echo 'hi42bye nice421423 bad42 cool_42a 42c' | sed ##### add your solution here
hi[42]bye nice[42]1[42]3 bad42 cool_[42]a 42c
```

c) Add `[]` around words starting with `s` and containing `e` and `t` in any order.

```
$ words='sequoia subtle exhibit asset sets tests site'
$ echo "$words" | sed ##### add your solution here
sequoia [subtle] exhibit asset [sets] tests [site]
```

d) Replace all whole words with `X` that start and end with the same word character.

```
$ echo 'oreo not a _a2_ roar took 22' | sed ##### add your solution here
X not X X X took X
```

e) Replace all occurrences of `[4]|*` with `2`

```
$ echo '2.3/[4]|*6 foo 5.3-[4]|*9' | sed ##### add your solution here
2.3/26 foo 5.3-29
```

f) `sed -nE '/\b[a-z](on|no)[a-z]\b/p'` is same as `sed -nE '/\b[a-z][on]{2}[a-z]\b/p'` . True or False? Sample input shown below might help to understand the differences, if any.

```
$ printf 'known\nmood\nknow\npony\ninns\n'
known
mood
know
pony
inns
```

g) Print all lines that start with `hand` and ends with no further character or `s` or `y` or `le` .

```
$ lines='handed\nhand\nhandy\nunhand\nhands\nhandle\n'
$ printf '%b' "$lines" | sed ##### add your solution here
hand
handy
hands
handle
```

h) Replace `42//5` or `42/5` with `8` for the given input.

```
$ echo 'a+42//5-c pressure*3+42/5-14256' | sed ##### add your solution here
a+8-c pressure*3+8-14256
```

i) For the given quantifiers, what would be the equivalent form using `{m,n}` representation?

- `?` is same as
- `*` is same as
- `+` is same as

j) True or False? In ERE, `(a*|b*)` is same as `(a|b)*`

k) For the given input, construct two different REGEXPs to get the outputs as shown below.

```
$ # delete from '(' till next ')'
$ echo 'a/b(division) + c%d() - (a#(b)2(' | sed ##### add your solution here
a/b + c%d - 2(

$ # delete from '(' till next ')' but not if there is '(' in between
$ echo 'a/b(division) + c%d() - (a#(b)2(' | sed ##### add your solution here
a/b + c%d - (a#2(
```

l) For the input file `anchors.txt` , convert **markdown** anchors to corresponding **hyperlinks**.

```
$ cat anchors.txt
# <a name="regular-expressions"></a>Regular Expressions
## <a name="subexpression-calls"></a>Subexpression calls
```

```
## <a name="the-dot-meta-character"></a>The dot meta character

$ sed ##### add your solution here
[Regular Expressions](#regular-expressions)
[Subexpression calls](#subexpression-calls)
[The dot meta character](#the-dot-meta-character)
```

m) Replace the space character that occurs after a word ending with `a` or `r` with a newline character.

```
$ echo 'area not a _a2_ roar took 22' | sed ##### add your solution here
area
not a
_a2_ roar
took 22
```

n) Surround all whole words with `()`. Additionally, if the whole word is `imp` or `ant`, delete them. Can you do it with single substitution?

```
$ words='tiger imp goat eagle ant important'
$ echo "$words" | sed ##### add your solution here
(tiger) () (goat) (eagle) () (important)
```

Flags

Just like options change the default behavior of shell commands, flags are used to change aspects of regular expressions. Some of the flags like `g` and `p` have been already discussed. For completeness, they will be discussed again in this chapter. In regular expression parlance, flags are also known as **modifiers**.

Case insensitive matching

The `I` flag allows to match a pattern case insensitively.

```
$ # match 'cat' case sensitively
$ printf 'Cat\ncOnCaT\nscatter\ncot\n' | sed -n '/cat/p'
scatter

$ # match 'cat' case insensitively
$ # note that command p cannot be used before flag I
$ printf 'Cat\ncOnCaT\nscatter\ncot\n' | sed -n '/cat/Ip'
Cat
cOnCaT
scatter

$ # match 'cat' case insensitively and replace it with 'dog'
$ printf 'Cat\ncOnCaT\nscatter\ncot\n' | sed 's/cat/dog/I'
dog
cOndog
sdogter
cot
```

Usually `i` is used for such purposes, `grep -i` for example. But `i` is a command (discussed in append, change, insert chapter) in `sed` , so `/REGEXP/i` cannot be used. The substitute command does allow both `i` and `I` to be used, but `I` is recommended for consistency.

Changing case in replacement section

This section isn't actually about flags, but presented in this chapter to complement the `I` flag. `sed` provides escape sequences to change the case of replacement strings, which might include backreferences, shell variables, etc.

Escape Sequence	Description
`\E`	indicates end of case conversion
`\l`	convert next character to lowercase
`\u`	convert next character to uppercase
`\L`	convert following characters to lowercase, unless `\U` or `\E` is used
`\U`	convert following characters to uppercase, unless `\L` or `\E` is used

First up, changing case of only the immediate next character after the escape sequence.

```
$ # match only first character of word using word boundary
$ # use & to backreference the matched character
$ # \u would then change it to uppercase
$ echo 'hello there. how are you?' | sed 's/\b\w/\u&/g'
Hello There. How Are You?

$ # change first character of word to lowercase
$ echo 'HELLO THERE. HOW ARE YOU?' | sed 's/\b\w/\l&/g'
hELLO tHERE. hOW aRE yOU?

$ # match lowercase followed by underscore followed by lowercase
$ # delete underscore and convert 2nd lowercase to uppercase
$ echo '_foo aug_price next_line' | sed -E 's/([a-z])_([a-z])/\1\u\2/g'
_foo augPrice nextLine
```

Next, changing case of multiple characters at a time.

```
$ # change all alphabets to lowercase
$ echo 'HaVE a nICe dAy' | sed 's/.*/\L&/'
have a nice day
$ # change all alphabets to uppercase
$ echo 'HaVE a nICe dAy' | sed 's/.*/\U&/'
HAVE A NICE DAY

$ # \E will stop further conversion
$ echo '_foo aug_price next_line' | sed -E 's/([a-z]+)(_[a-z]+)/\U\1\E\2/g'
_foo AUG_price NEXT_line
$ # \L or \U will override any existing conversion
$ echo 'HeLLo:bYe gOoD:beTTEr' | sed -E 's/([a-z]+)(:[a-z]+)/\L\1\U\2/Ig'
hello:BYE good:BETTER
```

Finally, examples where escapes can be used next to each other.

```
$ # uppercase first character of a word
$ # and lowercase rest of the word characters
$ # note the order of escapes used, \u\L won't work
$ echo 'HeLLo:bYe gOoD:beTTEr' | sed -E 's/[a-z]+/\L\u&/Ig'
Hello:Bye Good:Better

$ # lowercase first character of a word
$ # and uppercase rest of the word characters
$ echo 'HeLLo:bYe gOoD:beTTEr' | sed -E 's/[a-z]+/\U\l&/Ig'
hELLO:bYE gOOD:bETTER
```

Global replace

As seen earlier, by default substitute command will replace only the first occurrence of search pattern. Use `g` flag to replace all the matches.

```
$ # change only first ',' to '-'
$ printf '1,2,3,4\na,b,c,d\n' | sed 's/,/-/'
1-2,3,4
a-b,c,d

$ # change all matches by adding 'g' flag
$ printf '1,2,3,4\na,b,c,d\n' | sed 's/,/-/g'
1-2-3-4
a-b-c-d
```

Replace specific occurrences

A number provided as a flag will cause only the *N*th match to be replaced.

```
$ # default substitution replaces first occurrence
$ echo 'foo:123:bar:baz' | sed 's/:/-/'
foo-123:bar:baz
$ echo 'foo:123:bar:baz' | sed -E 's/[^:]+/"&"/'
"foo":123:bar:baz

$ # replace second occurrence
$ echo 'foo:123:bar:baz' | sed 's/:/-/2'
foo:123-bar:baz
$ echo 'foo:123:bar:baz' | sed -E 's/[^:]+/"&"/2'
foo:"123":bar:baz

$ # replace third occurrence and so on
$ echo 'foo:123:bar:baz' | sed 's/:/-/3'
foo:123:bar-baz
$ echo 'foo:123:bar:baz' | sed -E 's/[^:]+/"&"/3'
foo:123:"bar":baz
```

Quantifiers can be used to replace *N*th match from the end of line.

```
$ # replacing last occurrence
$ # can also use sed -E 's/:([^:]*)$/[]\1/'
$ echo '456:foo:123:bar:789:baz' | sed -E 's/(.*):/\1[]/'
456:foo:123:bar:789[]baz

$ # replacing last but one
$ echo '456:foo:123:bar:789:baz' | sed -E 's/(.*):(.*:)/\1[]\2/'
456:foo:123:bar[]789:baz

$ # generic version, where {N} refers to last but N
$ echo '456:foo:123:bar:789:baz' | sed -E 's/(.*):((.*:){2})/\1[]\2/'
456:foo:123[]bar:789:baz
```

See unix.stackexchange: Why doesn't this sed command replace the 3rd-to-last "and"? for a bug related to use of word boundaries in the `(){N}` generic case.

A combination of number and `g` flag will replace all matches except the first *N-1* occurrences. In other words, all matches starting from the *N*th occurrence will be replaced.

```
$ # replace all except the first occurrence
$ echo '456:foo:123:bar:789:baz' | sed -E 's/:/[]/2g'
456:foo[]123[]bar[]789[]baz

$ # replace all except the first three occurrences
$ echo '456:foo:123:bar:789:baz' | sed -E 's/:/[]/4g'
456:foo:123:bar[]789[]baz
```

If multiple *N*th occurrences are to be replaced, use descending order for readability.

```
$ # replace second and third occurrences
$ # note the numbers used
$ echo '456:foo:123:bar:789:baz' | sed 's/:/[]/2; s/:/[]/2'
456:foo[]123[]bar:789:baz

$ # better way is to use descending order
$ echo '456:foo:123:bar:789:baz' | sed 's/:/[]/3; s/:/[]/2'
456:foo[]123[]bar:789:baz

$ # replace second, third and fifth occurrences
$ echo '456:foo:123:bar:789:baz' | sed 's/:/[]/5; s/:/[]/3; s/:/[]/2'
456:foo[]123[]bar:789[]baz
```

Print flag

This flag was already introduced in Selective editing chapter.

```
$ # no output if no substitution
$ echo 'hi there. have a nice day' | sed -n 's/xyz/XYZ/p'

$ # modified line is displayed if substitution succeeds
$ echo 'hi there. have a nice day' | sed -n 's/\bh/H/pg'
Hi there. Have a nice day
```

Write to a file

The `w` flag allows to redirect contents to a specified filename instead of default **stdout**. This flag applies to both filtering and substitution command. You might wonder why not simply use shell redirection? As `sed` allows multiple commands, the `w` flag can be used selectively, allow writes to multiple files and so on.

```
$ # space between w and filename is optional
$ # same as: sed -n 's/3/three/p' > 3.txt
$ seq 20 | sed -n 's/3/three/w 3.txt'
$ cat 3.txt
three
1three

$ # do not use -n if output should be displayed as well as written to file
$ printf '1,2,3,4\na,b,c,d\n' | sed 's/,/:/gw cols.txt'
1:2:3:4
a:b:c:d
$ cat cols.txt
1:2:3:4
a:b:c:d
```

For multiple output files, use `-e` for each file. Don't use `;` between commands as that will be interpreted as part of the filename!

```
$ seq 20 | sed -n -e 's/5/five/w 5.txt' -e 's/7/seven/w 7.txt'
$ cat 5.txt
five
1five
$ cat 7.txt
seven
1seven
```

There are two predefined filenames:

- `/dev/stdout` to write to **stdout**
- `/dev/stderr` to write to **stderr**

```
$ # in-place editing as well as display changes on stdout
$ sed -i 's/three/3/w /dev/stdout' 3.txt
3
13
$ cat 3.txt
3
13
```

Executing external commands

The `e` flag allows to use output of a shell command. The external command can be based on the pattern space contents or provided as an argument. Quoting from the manual:

> This command allows one to pipe input from a shell command into pattern space. Without parameters, the e command executes the command that is found in pattern space and replaces the pattern space with the output; a trailing newline is suppressed.

> If a parameter is specified, instead, the e command interprets it as a command and sends its output to the output stream. The command can run across multiple lines, all but the last ending with a back-slash.

> In both cases, the results are undefined if the command to be executed contains a NUL character.

First, examples with substitution command.

```
$ # sample input
$ printf 'Date:\nreplace this line\n'
Date:
replace this line

$ # replacing entire line with output of shell command
$ printf 'Date:\nreplace this line\n' | sed 's/^replace.*/date/e'
Date:
Wed Aug 14 11:39:39 IST 2019
```

If the `p` flag is used as well, order is important. Quoting from the manual:

> when both the p and e options are specified, the relative ordering of the two produces very different results. In general, ep (evaluate then print) is what you want, but operating the other way round can be useful for debugging. For this reason, the current version of GNU sed interprets specially the presence of p options both before and after e, printing the pattern space before and after evaluation, while in general flags for the s command show their effect just once. This behavior, although documented, might change in future versions.

```
$ printf 'Date:\nreplace this line\n' | sed -n 's/^replace.*/date/ep'
Wed Aug 14 11:42:48 IST 2019

$ printf 'Date:\nreplace this line\n' | sed -n 's/^replace.*/date/pe'
date
```

If only a portion of the line is replaced, complete modified line after substitution will get executed as a shell command.

```
$ # after substitution, the command that gets executed is 'seq 5'
$ echo 'xyz 5' | sed 's/xyz/seq/e'
1
2
3
4
5
```

Next, examples with filtering alone.

```
$ # execute entire matching line as a shell command
$ # replaces the matching line with output of the command
```

```
$ printf 'date\ndate -I\n' | sed '/date/e'
Wed Aug 14 11:51:06 IST 2019
2019-08-14
$ printf 'date\ndate -I\n' | sed '2e'
date
2019-08-14

$ # command provided as argument, output is inserted before matching line
$ printf 'show\nexample\n' | sed '/am/e seq 2'
show
1
2
example
```

Multiline mode

The `m` (or `M`) flag will change the behavior of `^`, `$` and `.` metacharacters. This comes into play only if there are multiple lines in the pattern space to operate with, for example when the `N` command is used.

If `m` flag is used, the `.` metacharacter will not match the newline character.

```
$ # without m flag . will match newline character
$ printf 'Hi there\nHave a Nice Day\n' | sed 'N; s/H.*e/X/'
X Day

$ # with m flag . will not match across lines
$ printf 'Hi there\nHave a Nice Day\n' | sed 'N; s/H.*e/X/gm'
X
X Day
```

The `^` and `$` anchors will match every line's start and end locations when `m` flag is used.

```
$ # without m flag line anchors will match once for whole string
$ printf 'Hi there\nHave a Nice Day\n' | sed 'N; s/^/* /g'
* Hi there
Have a Nice Day
$ printf 'Hi there\nHave a Nice Day\n' | sed 'N; s/$/./g'
Hi there
Have a Nice Day.

$ # with m flag line anchors will work for every line
$ printf 'Hi there\nHave a Nice Day\n' | sed 'N; s/^/* /gm'
* Hi there
* Have a Nice Day
$ printf 'Hi there\nHave a Nice Day\n' | sed 'N; s/$/./gm'
Hi there.
Have a Nice Day.
```

The `\`` and `\'` anchors will always match the start and end of entire string, irrespective

of single or multiline mode.

```
$ # similar to \A start of string anchor found in other implementations
$ printf 'Hi there\nHave a Nice Day\n' | sed 'N; s/\`/* /gm'
* Hi there
Have a Nice Day

$ # similar to \Z end of string anchor found in other implementations
$ # note the use of double quotes
$ # with single quotes, it will be: sed 'N; s/\'\''/./gm'
$ printf 'Hi there\nHave a Nice Day\n' | sed "N; s/\'/./gm"
Hi there
Have a Nice Day.
```

Usually, regular expression implementations have separate flags to control the behavior of `.` metacharacter and line anchors. Having a single flag restricts flexibility. As an example, you cannot make `.` to match across lines if `m` flag is used in `sed`. You'll have to resort to some creative alternatives in such cases as shown below.

```
$ # \w|\W or .|\n can also be used
$ # recall that sed doesn't allow character set sequences inside []
$ printf 'Hi there\nHave a Nice Day\n' | sed -E 'N; s/H(\s|\S)*e/X/m'
X Day

$ # this one doesn't use alternation
$ printf 'Hi there\nHave a Nice Day\n' | sed -E 'N; s/H(.*\n.*)*e/X/m'
X Day
```

Cheatsheet and summary

Note	Description
flag	changes default behavior of REGEXP
`I`	match case insensitively for REGEXP address
`i` or `I`	match case insensitively for substitution command
`\E`	indicates end of case conversion in replacement section
`\l`	convert next character to lowercase
`\u`	convert next character to uppercase
`\L`	convert following characters to lowercase, unless `\U` or `\E` is used
`\U`	convert following characters to uppercase, unless `\L` or `\E` is used
`g`	replace all occurrences instead of just the first match
`N`	a number will cause only the *N*th match to be replaced
`p`	prints line only if substitution succeeds (assuming `-n` is active)
`w filename`	write contents of pattern space to given filename whenever the REGEXP address matches or substitution succeeds
`e`	executes contents of pattern space as shell command and replaces the pattern space with command output if argument is passed, executes that external command and inserts output before matching lines
`m` or `M`	multiline mode flag

Note	Description
	`.` will not match the newline character
	`^` and `$` will match every line's start and end locations
`\``	always match the start of string irrespective of `m` flag
`\'`	always match the end of string irrespective of `m` flag

This chapter showed how flags can be used for extra functionality. Some of the flags interact with the shell as well. In the next chapter, you'll learn how to incorporate shell variables and command outputs to dynamically construct a `sed` command.

Exercises

a) For the input file `para.txt`, remove all groups of lines marked with a line beginning with `start` and a line ending with `end`. Match both these markers case insensitively.

```
$ cat para.txt
good start
Start working on that
project you always wanted
to, do not let it end
hi there
start and try to
finish the End
bye

$ sed ##### add your solution here
good start
hi there
bye
```

b) The given sample input below starts with one or more `#` characters followed by one or more whitespace characters and then some words. Convert such strings to corresponding output as shown below.

```
$ echo '# Regular Expressions' | sed ##### add your solution here
regular-expressions
$ echo '## Compiling regular expressions' | sed ##### add your solution here
compiling-regular-expressions
```

c) Using the input file `para.txt`, create a file named `five.txt` with all lines that contain a whole word of length **5** and a file named `six.txt` with all lines that contain a whole word of length **6**.

```
$ sed ##### add your solution here

$ cat five.txt
good start
Start working on that
hi there
```

```
start and try to
$ cat six.txt
project you always wanted
finish the End
```

d) Given sample strings have fields separated by `,` where field values can be empty as well. Use `sed` to replace the third field with `42` .

```
$ echo 'lion,,ant,road,neon' | sed ##### add your solution here
lion,,42,road,neon

$ echo ',,,' | sed ##### add your solution here
,,42,
```

e) Replace all occurrences of `e` with `3` except the first two matches.

```
$ echo 'asset sets tests site' | sed ##### add your solution here
asset sets t3sts sit3

$ echo 'sample item teem eel' | sed ##### add your solution here
sample item t33m 33l
```

f) For the input file `addr.txt` , replace all input lines with number of characters in those lines. `wc -L` is one of the ways to get length of a line as shown below.

```
$ # note that newline character isn't counted, which is preferable here
$ echo "Hello World" | wc -L
11

$ sed ##### add your solution here
11
11
17
14
5
13
```

g) For the input file `para.txt` , assume that it'll always have lines in multiples of 4. Use `sed` commands such that there are 4 lines at a time in the pattern space. Then, delete from `start` till `end` provided `start` is matched only at the start of a line. Also, match these two keywords case insensitively.

```
$ sed ##### add your solution here
good start

hi there

bye
```

h) For the given strings, replace last but third `so` with `X` . Only print the lines which are changed by the substitution.

```
$ printf 'so and so also sow and soup\n' | sed ##### add your solution here
so and X also sow and soup

$ printf 'sososososososo\nso and so\n' | sed ##### add your solution here
sososoXsososo
```

i) Display all lines that satisfies **both** of these conditions:

- `professor` matched irrespective of case
- `quip` or `this` matched case sensitively

Input is a file downloaded from internet as shown below.

```
$ wget https://www.gutenberg.org/files/345/345.txt -O dracula.txt

$ sed ##### add your solution here
equipment of a professor of the healing craft. When we were shown in,
should be. I could see that the Professor had carried out in this room,
"Not up to this moment, Professor," she said impulsively, "but up to
and sprang at us. But by this time the Professor had gained his feet,
this time the Professor had to ask her questions, and to ask them pretty
```

Shell substitutions

So far, the `sed` commands have been constructed statically. All the details were known. For example, which line numbers to act upon, the search REGEXP, the replacement string and so on. When it comes to automation and scripting, you'd often need to construct commands dynamically based on user input, file contents, etc. And sometimes, output of a shell command is needed as part of the replacement string. This chapter will discuss how to incorporate shell variables and command output to compose a `sed` command dynamically. As mentioned before, this book assumes `bash` as the shell being used.

As an example, see my repo ch: command help for a practical shell script, where commands are constructed dynamically.

Variable substitution

The characters you type on the command line are first interpreted by the shell before it can be executed. Wildcards are expanded, pipes and redirections are set up, double quotes are interpolated and so on. For some use cases, it is simply easier to use double quotes instead of single quotes for the script passed to `sed` command. That way, shell variables get substituted for their values.

See wooledge: Quotes and unix.stackexchange: Why does my shell script choke on whitespace or other special characters? for details about various quoting mechanisms in `bash` and when quoting is needed.

```
$ start=5; step=1
$ sed -n "${start},+${step}p" programming_quotes.txt
Some people, when confronted with a problem, think - I know, I will
use regular expressions. Now they have two problems by Jamie Zawinski

$ step=4
$ sed -n "${start},+${step}p" programming_quotes.txt
Some people, when confronted with a problem, think - I know, I will
use regular expressions. Now they have two problems by Jamie Zawinski

A language that does not affect the way you think about programming,
is not worth knowing by Alan Perlis
```

But, if the shell variables can contain any generic string instead of just numbers, it is strongly recommended to use double quotes only where it is needed. Otherwise, normal characters part of `sed` command may get interpolated because of double quotes. `bash` allows unquoted, single quoted and double quoted strings to be concatenated by simply placing them next to each other.

```
$ # ! is special within double quotes
$ # !d got expanded to 'date -Is' from my history and hence the error
```

```
$ word='at'
$ printf 'sea\neat\ndrop\n' | sed "/${word}/!d"
printf 'sea\neat\ndrop\n' | sed "/${word}/date -Is"
sed: -e expression #1, char 6: extra characters after command

$ # use double quotes only for variable substitution
$ # and single quotes for everything else
$ # the command is concatenation of '/' and "${word}" and '/!d'
$ printf 'sea\neat\ndrop\n' | sed '/'"${word}"'/!d'
eat
```

After you've properly separated single and double quoted portions, you need to take care of few more things to robustly construct a dynamic command. First, you'll have to ensure that the shell variable is properly preprocessed to avoid conflict with whichever delimiter is being used for search or substitution operations.

See wooledge: Parameter Expansion for details about the `bash` feature used in the example below.

```
$ # error because '/' inside HOME value conflicts with '/' as delimiter
$ echo 'home path is:' | sed 's/$/ '"${HOME}"'/'
sed: -e expression #1, char 7: unknown option to `s'
$ # using a different delimiter will help in this particular case
$ echo 'home path is:' | sed 's|$| '"${HOME}"'|'
home path is: /home/learnbyexample

$ # but you may not have the luxury of choosing a delimiter
$ # in such cases, escape all delimiter characters before variable substitution
$ home=${HOME//\//\\/}
$ echo 'home path is:' | sed 's/$/ '"${home}"'/'
home path is: /home/learnbyexample
```

If the variable value is obtained from an external source, such as user input, then you need to worry about security too. See unix.stackexchange: security consideration when using shell substitution for more details.

Escaping metacharacters

Next, you have to properly escape all the metacharacters depending upon whether the variable is used as search or replacement string. This is needed only if the content of the variable has to be treated literally. Here's an example to illustrate the issue with one metacharacter.

```
$ c='&'
$ # & will backreference entire matched portion
$ echo 'a and b and c' | sed 's/and/'"${c}"'/g'
a [and] b [and] c
```

```
$ # escape all occurrences of & to insert it literally
$ c1=${c//&/\\&}
$ echo 'a and b and c' | sed 's/and/'"${c1}"'/g'
a [&] b [&] c
```

Typically, you'd need to escape `\` , `&` and the delimiter for variables used in the replacement section. For the search section, the characters to be escaped will depend upon whether you are using BRE or ERE.

```
$ # replacement string
$ r='a/b&c\d'
$ r=$(printf '%s' "$r" | sed 's#[\&/]#\\&#g')

$ # ERE version for search string
$ s='{[(\ta^b/d).*+?^$|]}'
$ s=$(printf '%s' "$s" | sed 's#[{[()^$*?+.\|/]#\\&#g')
$ echo 'f*{[(\ta^b/d).*+?^$|]} - 3' | sed -E 's/'"$s"'/'"$r"'/g'
f*a/b&c\d - 3

$ # BRE version for search string
$ s='{[(\ta^b/d).*+?^$|]}'
$ s=$(printf '%s' "$s" | sed 's#[[^$*.\/]#\\&#g')
$ echo 'f*{[(\ta^b/d).*+?^$|]} - 3' | sed 's/'"$s"'/'"$r"'/g'
f*a/b&c\d - 3
```

For a more detailed analysis on escaping the metacharacters, refer to these wonderful Q&A threads.

- unix.stackexchange: How to ensure that string interpolated into sed substitution escapes all metacharacters — also discusses how to escape literal newlines in replacement string
- stackoverflow: Is it possible to escape regex metacharacters reliably with sed
- unix.stackexchange: What characters do I need to escape when using sed in a script?

Command substitution

This section will show examples of using output of shell command as part of `sed` command. And all the precautions seen in previous sections apply here too.

See also wooledge: Why is $() preferred over backticks?

```
$ # note that the trailing newline character of command output gets stripped
$ echo 'today is date.' | sed 's/date/'"$(date -I)"'/'
today is 2019-08-23.

$ # need to change delimiter where possible
$ printf 'f1.txt\nf2.txt\n' | sed 's|^|'"$(pwd)"'/|'
/home/learnbyexample/f1.txt
/home/learnbyexample/f2.txt
```

```
$ # or preprocess if delimiter cannot be changed for other reasons
$ p=$(pwd | sed 's|/|\\/|g')
$ printf 'f1.txt\nf2.txt\n' | sed 's/^/'"${p}"'\//'
/home/learnbyexample/f1.txt
/home/learnbyexample/f2.txt
```

Multiline command output cannot be substituted in this manner, as substitute command doesn't allow literal newlines in replacement section unless escaped.

```
$ printf 'a\n[x]\nb\n' | sed 's/x/'"$(seq 3)"'/'
sed: -e expression #1, char 5: unterminated `s' command
$ # prefix literal newlines with \ except the last newline
$ printf 'a\n[x]\nb\n' | sed 's/x/'"$(seq 3 | sed '$!s/$/\\/' )"'/'
a
[1
2
3]
b
```

Cheatsheet and summary

Note	Description
`sed -n "${start},+${step}p"`	dynamically construct `sed` command in above example, `start` and `step` are shell variables their values gets substituted before `sed` is executed
`sed "/${word}/!d"`	entire command in double quotes is risky within double quotes, `$` , `\` , `!` and `` ` `` are special
`sed '/'"${word}"'/!d'`	use double quotes only where needed and variable contents have to be preprocessed to prevent clashing with `sed` metacharacters and security issue if you don't control the variable contents
`sed 's#[\&/]#\\&#g'`	escape metacharacters for replacement section
`sed '$!s/$/\\/'`	escape literal newlines for replacement section
`sed 's#[{[()^$*?+.|/]#\\&#g'`	escape metacharacters for search section, ERE
`sed 's#[[^$*.\/]#\\&#g'`	escape metacharacters for search section, BRE
`sed 's/date/'"$(date -I)"'/'`	example for command substitution command output's final newline character gets stripped other literal newlines, if any, have to be escaped

This chapter covered some of the ways to construct a `sed` command dynamically. Like most things in software programming, 90% of the cases are relatively easier to accomplish. But the other 10% could get significantly complicated. Dealing with the clash between shell and `sed` metacharacters is a mess and I'd even suggest looking for alternatives such as `perl` to reduce the complexity. The next chapter will cover some more command line options.

Exercises

a) Replace `#expr#` with value of `usr_ip` shell variable. Assume that this variable can only contain the metacharacters as shown in the sample below.

```
$ usr_ip='c = (a/b) && (x-5)'
$ mod_ip=$(echo "$usr_ip" | sed ##### add your solution here)
$ echo 'Expression: #expr#' | sed ##### add your solution here
Expression: c = (a/b) && (x-5)
```

b) Repeat previous exercise, but this time with command substitution instead of using temporary variable.

```
$ usr_ip='c = (a/b/y) && (x-5)'
$ echo 'Expression: #expr#' | sed ##### add your solution here
Expression: c = (a/b/y) && (x-5)
```

z, s and f command line options

This chapter covers the `-z`, `-s` and `-f` command line options. These come in handy for specific use cases. For example, the `-z` option helps to process input separated by ASCII NUL character and the `-f` option allows you to pass `sed` commands from a file.

NUL separated lines

The `-z` option will cause `sed` to separate lines based on the ASCII NUL character instead of the newline character. Just like normal newline based processing, the NUL character is removed (if present) from the input line and added back accordingly when the processed line is printed.

NUL separation is very useful to process filenames as newline is a valid character for filenames but NUL character isn't. For example, `grep -Z` and `find -print0` will print NUL separated filenames whereas `grep -z` and `xargs -0` will accept NUL separated input.

```
$ printf 'earn xerox\0at\nmare\npart\0learn eye\n' | sed -nz '/are/p'
at
mare
part

$ # \0 at end of output depends on input having \0 character
$ printf 'earn xerox\0at\nmare\npart\0learn eye\n' | sed -nz '/are/p' | od -c
0000000   a   t  \n   m   a   r   e  \n   p   a   r   t  \0
0000015
$ printf 'earn xerox\0at\nmare\npart\0learn eye\n' | sed -nz '/eye/p' | od -c
0000000   l   e   a   r   n       e   y   e  \n
0000012
```

If input doesn't have NUL characters, then `-z` option is handy to process the entire input as a single string. This is effective only for files small enough to fit your available machine memory. It would also depend on the regular expression, as some patterns have exponential relationship with respect to data size. As input doesn't have NUL character, output also wouldn't have NUL character, unlike `GNU grep` which always adds the line separator to the output.

```
$ # adds ; to previous line if current line starts with c
$ printf 'cater\ndog\ncoat\ncutter\nmat\n' | sed -z 's/\nc/;&/g'
cater
dog;
coat;
cutter
mat
```

Separate files

The `-s` option will cause `sed` to treat multiple input files separately instead of treating them as single concatenated input. This helps if you need line number addressing to be effective for each input file.

If `-i` option is being used, `-s` is also implied.

```
$ # without -s, there is only one first line for all files combined
$ # F command inserts filename of current file at the given address
$ sed '1F' cols.txt 5.txt
cols.txt
1:2:3:4
a:b:c:d
five
1five

$ # with -s, each file has its own address
$ # this is like 'cat' command but also prints filename as header
$ sed -s '1F' cols.txt 5.txt
cols.txt
1:2:3:4
a:b:c:d
5.txt
five
1five
```

File as source of sed commands

If your `sed` commands does not easily fit the command line, you have the option of putting the commands in a file and use `-f` option to specify that file as the source of commands to execute. This method also provides benefits like:

- literal newline can be used to separate the commands
- single quotes can be freely used as it will no longer clash as a shell metacharacter
- comments can be specified after the `#` character
    - See also sed manual: Often-Used Commands for details about using comments

Consider the following input file and a `sed` script written into a plain text file.

```
$ cat sample.txt
Hi there
cats and dogs running around
Have a nice day

$ cat word_mapping.sed
# word mappings
s/cat/Cat/g
s/dog/Dog/g
```

```
s/Hi/Hey/g
# appending another line
/there|running/ s/$/\n----------/
```

The two lines starting with `#` character are comment lines. Comments can also be added at end of a command line if required, just like other programming languages. Use the `-f` option to pass the contents of the file as `sed` commands. Any other command line option like `-n` , `-z` , `-E` , etc have to mentioned along with `sed` invocation, just like you've done so far.

```
$ # the first 3 substitutions will work
$ # but not the last one, as | is not a metacharacter with BRE
$ sed -f word_mapping.sed sample.txt
Hey there
Cats and Dogs running around
Have a nice day

$ # | now works as ERE is enabled using -E option
$ sed -E -f word_mapping.sed sample.txt
Hey there
----------
Cats and Dogs running around
----------
Have a nice day
```

Similar to making an executable `bash` or `perl` or `python` script, you can add a shebang (see wikipedia: shebang for details) line to a `sed` script file.

```
$ # to get full path of the command
$ type sed
sed is /usr/local/bin/sed

$ # sed script with shebang, note the use of -f after command path
$ cat executable.sed
#!/usr/local/bin/sed -f
s/cats\|dogs/'&'/g

$ # add execute permission to the script
$ chmod +x executable.sed

$ # executing the script
$ ./executable.sed sample.txt
Hi there
'cats' and 'dogs' running around
Have a nice day
```

Adding any other command line option like `-n` , `-z` , `-E` , etc depends on a lot of factors. See stackoverflow: usage of options along with shebang for details.

See also sed manual: Some Sample Scripts and Sokoban game written in sed

Cheatsheet and summary

Note	Description
`-z`	change line separator from newline to ASCII NUL character `grep -Z` and `find -print0` are examples for NUL separated input `-z` also useful to process entire input as single string if it doesn't contain NUL characters, assuming small enough input file to fit memory
`-s`	separate addressing for multiple file inputs `-s` is implied if `-i` is being used
`-f`	supply commands from a file literal newline can be used to separate the commands single quotes can be freely used comments can be specified after the `#` character
`F`	command to insert current filename at the given address

This chapter covered three command line options that come in handy for specific situations. You also saw a few examples of `sed` being used as part of a solution with other commands in a pipeline or a shell script. In the next chapter, you'll learn three commands that are also specialized for particular use cases.

Exercises

a) Replace any character other than word characters and `.` character with `_` character for the sample filenames shown below.

```
$ mkdir test_dir && cd $_
$ touch 'file with spaces.txt' $'weird$ch\nars.txt' '!f@oo.txt'
$ # > at start of line indicates continuation of multiline shell command
$ for file in *; do
>   new_name=$(printf '%s' "$file" | sed ##### add your solution here)
>   mv "$file" "$new_name"
> done

$ ls
file_with_spaces.txt  _f_oo.txt  weird_ch_ars.txt
$ cd .. && rm -r test_dir
```

b) Print only the third line, if any, from these input files: `addr.txt`, `para.txt` and `copyright.txt`

```
$ sed ##### add your solution here
This game is good
```

```
project you always wanted
bla bla bla
```

c) For the input file `hex.txt` , use content from `replace.txt` to perform search and replace operations. Each line in `replace.txt` starts with the search term, followed by a space and then followed by the replace term. Assume that these terms do not contain any `sed` metacharacters.

```
$ cat hex.txt
start address: 0xA0, func1 address: 0xA0
end address: 0xFF, func2 address: 0xB0
$ cat replace.txt
0xA0 0x5000
0xB0 0x6000
0xFF 0x7000

$ sed -f <(sed ##### add your solution here) hex.txt
start address: 0x5000, func1 address: 0x5000
end address: 0x7000, func2 address: 0x6000
```

append, change, insert

These three commands come in handy for specific operations as suggested by their names. The substitute command could handle most of the features offered by these commands. But where applicable, these commands would be easier to use.

Unless otherwise specified, rules mentioned in following sections will apply similarly for all the three commands.

Basic usage

Just like the substitute command, first letter of these three names represents the command in a `sed` script.

- `a` appends given string after end of line of each of the matching address
- `c` changes the entire matching address contents to the given string
- `i` inserts given string before start of line of each of the matching address

The string value for these commands is supplied after the command letter. Any whitespace between the letter and the string value is ignored. First up, some examples with single address as the qualifier.

```
$ # same as: sed '2 s/$/\nhello/'
$ seq 3 | sed '2a hello'
1
2
hello
3

$ # same as: sed '/[24]/ s/.*/hello/'
$ seq 5 | sed '/[24]/c hello'
1
hello
3
hello
5

$ # same as: sed '2 s/^/hello\n/'
$ seq 3 | sed '2i hello'
1
hello
2
3
```

Next, examples with address ranges.

```
$ # append and insert will apply for each matching line of address range
$ seq 5 | sed '2,4i hi there!'
1
```

```
hi there!
2
hi there!
3
hi there!
4
5

$ # change will replace entire matching range with given string
$ seq 5 | sed '2,4c hi there!'
1
hi there!
5
$ # to change every matching line, use substitute command
$ seq 5 | sed '2,4 s/.*/hi there!/'
1
hi there!
hi there!
hi there!
5
```

Escape sequences

Similar to replacement strings in substitute command, you can use escape sequences like `\t` , `\n` , etc and ASCII value formats like `\xNN` .

```
$ seq 3 | sed '2c rat\tdog\nwolf'
1
rat     dog
wolf
3
$ seq 3 | sed '2a it\x27s sunny today'
1
2
it's sunny today
3
```

As mentioned before, any whitespace between the command and the string is ignored. You can use `\` after the command letter to prevent that.

```
$ seq 3 | sed '2c         hello'
1
hello
3

$ seq 3 | sed '2c\         hello'
1
         hello
3
```

As `\` has another meaning when used immediately after command letter, use an additional `\` if there is a normal escape sequence at start of the string.

```
$ seq 3 | sed '2c\nhi'
1
nhi
3

$ seq 3 | sed '2c\\nhi'
1

hi
3
```

See also stackoverflow: add newline character if last line of input doesn't have one

Multiple commands

All the three commands will treat everything after the command letter as the string argument. Thus, you cannot use `;` as command separator or `#` to start a comment. Even command grouping with `{}` will fail unless you use `-e` option or literal newline to separate the closing `}`.

```
$ # 'hi ; 3a bye' will treated as single string argument
$ seq 4 | sed '2c hi ; 3a bye'
1
hi ; 3a bye
3
4
$ # } gets treated as part of argument for append command, hence the error
$ seq 3 | sed '2{s/^/*/; a hi}'
sed: -e expression #1, char 0: unmatched `{'

$ # use -e or literal newline to separate the commands
$ seq 4 | sed -e '2c hi' -e '3a bye'
1
hi
3
bye
4
$ seq 3 | sed '2{c hi
> }'
1
hi
3
```

Shell substitution

This section is included in this chapter to showcase more examples for shell substitutions and to warn about the potential pitfalls.

```
$ # variable substitution
$ text='good\tone\nfood\tpun'
$ seq 13 15 | sed '2c'"$text"
13
good    one
food    pun
15

$ # command substitution
$ seq 13 15 | sed '3i'"$(date +%A)"
13
14
Wednesday
15
```

Literal newline in the substituted string may cause an error depending upon content. To avoid the behavior shown below, process the command output as discussed in Command substitution section.

```
$ seq 13 15 | sed '3i'"$(printf 'hi\n123')"
sed: -e expression #1, char 8: missing command

$ # here, the content after newline gets executed as a command
$ # same as: sed -e '3i hi' -e 's/5/five/'
$ seq 13 15 | sed '3i'"$(printf 'hi\ns/5/five/')"
13
14
hi
1five
```

Cheatsheet and summary

Note	Description
`a`	appends given string after end of line of each of the matching address
`c`	changes the entire matching address contents to the given string
`i`	inserts given string before start of line of each of the matching address
	string value for these commands is supplied after the command letter
	escape sequences like `\t` , `\n` , `\xNN` , etc can be used in the string value
	any whitespace between command letter and the string value is ignored unless `\` is used after command letter
	`\` after command letter is also needed if escape sequence is the first character
	`-e` or literal newline is needed to separate any further commands

This chapter covered three more `sed` commands that work similarly to substitution command for specific use cases. The string argument to these commands allow escape sequences to be used. If you do not wish the text to be interpreted or if you wish to provide text from a file, then use the commands covered in next chapter, which allows you to add text literally.

Exercises

a) For the input file `addr.txt`, print only the third line and surround it with `-----`

```
$ sed ##### add your solution here
-----
This game is good
-----
```

b) For the input file `addr.txt`, replace all lines starting from a line containing `you` till end of file with content as shown below.

```
$ sed ##### add your solution here
Hello World

Have a nice day
```

c) Replace every even numbered line with `---`

```
$ seq 0 5 | sed ##### add your solution here
0
---
2
---
4
---
```

Adding content from file

The previous chapter discussed how to use `a` , `c` and `i` commands to append, change or insert the given string for matching address. Any `\` in the string argument is treated according to `sed` escape sequence rules and it cannot contain literal newline character. The `r` and `R` commands allow to use file contents as the source string which is always treated literally and can contain newline characters. Thus, these two commands provide a robust way to add multiline text literally.

However, `r` and `R` provide only *append* functionality for matching address. Other `sed` features will be used to show examples for `c` and `i` variations.

r for entire file

The `r` command accepts a filename as argument and when the address is satisfied, entire contents of the given file is added after the matching line.

If the given filename doesn't exist, `sed` will silently ignore it and proceed as if the file was empty. Exit status will be `0` unless something else goes wrong with the `sed` command used.

```
$ cat ip.txt
    * sky
    * apple
$ cat fav_colors.txt
deep red
yellow
reddish
brown

$ # space between r and filename is optional
$ # adds entire contents of 'ip.txt' after each line containing 'red'
$ sed '/red/r ip.txt' fav_colors.txt
deep red
    * sky
    * apple
yellow
reddish
    * sky
    * apple
brown
```

To use command output as file source, use `/dev/stdin` as filename.

```
$ text='good\tone\nfood\tpun'
$ echo "$text" | sed '1r /dev/stdin' ip.txt
    * sky
good\tone\nfood\tpun
    * apple
```

```
$ # example for adding multiline command output
$ seq 2 | sed '2r /dev/stdin' ip.txt
    * sky
    * apple
1
2

$ # note that newline won't be added to file contents being read
$ printf '123' | sed '1r /dev/stdin' ip.txt
    * sky
123    * apple
```

Here's some examples to emulate `c` command functionality with `r` command. Similar to `a` , `c` and `i` commands, everything after `r` and `R` commands is treated as filename. So, use `-e` or literal newlines when multiple commands are needed.

See also unix.stackexchange: Various ways to replace line M in file1 with line N in file2

```
$ # replacing only the 2nd line
$ # order is important, first 'r' and then 'd'
$ # note the use of command grouping to avoid repeating the address
$ items='    * blue\n    * green\n'
$ printf '%b' "$items" | sed -e '2 {r /dev/stdin' -e 'd}' ip.txt
    * sky
    * blue
    * green

$ # replacing range of lines
$ # using grouping here will add file contents for each matching line
$ # so, use 'r' only for second address
$ # and then delete the range, // here avoids duplicating second address
$ sed -e '/^red/r ip.txt' -e '/yellow/,//d' fav_colors.txt
deep red
    * sky
    * apple
brown
```

Quoting from manual:

> The empty regular expression '//' repeats the last regular expression match (the same holds if the empty regular expression is passed to the s command). Note that modifiers to regular expressions are evaluated when the regular expression is compiled, thus it is invalid to specify them together with the empty regular expression

Emulating `i` command functionality with `r` command requires advanced usage of `sed` and well beyond the scope of this book. See unix.stackexchange: insert file contents before matching line for examples. Instead of `r` command, the next section will show how to use

the `e` flag seen earlier for this purpose.

Using e and cat command

The manual has this handy note for the `e` flag:

> Note that, unlike the r command, the output of the command will be printed immediately; the r command instead delays the output to the end of the current cycle.

This makes the `e` flag the easiest way to insert file contents before the matching lines. Similar to `r` command, the output of external command is inserted literally. But one difference from `r` command is that if the filename passed to the external `cat` command doesn't exist, then you will see its error message inserted.

```
$ sed '/red/e cat ip.txt' fav_colors.txt
    * sky
    * apple
deep red
yellow
    * sky
    * apple
reddish
brown

$ text='good\tone\nfood\tpun'
$ echo "$text" | sed '1e cat /dev/stdin' ip.txt
good\tone\nfood\tpun
    * sky
    * apple
```

R for line by line

The `R` command is very similar to `r` with respect to most of the rules seen in previous section. But instead of reading entire file contents, `R` will read one line at a time from the source file when the given address matches. If entire file has already been read and another address matches, `sed` will proceed as if the line was empty.

```
$ sed '/red/R ip.txt' fav_colors.txt
deep red
    * sky
yellow
reddish
    * apple
brown

$ # interleave contents of two files
$ seq 4 | sed 'R /dev/stdin' fav_colors.txt
deep red
```

```
1
yellow
2
reddish
3
brown
4
```

See also stackoverflow: Replace first few lines with first few lines from other file

Cheatsheet and summary

Note	Description
`r filename`	entire contents of file is added after each matching line
`e cat filename`	entire contents of file is added before each matching line
`R filename`	add one line at a time from file after each matching line
	space between command and filename is optional
	use `/dev/stdin` as filename to use stdin as file source
	file contents are added literally, no escape sequence interpretation

This chapter covered powerful and robust solutions for adding text literally from a file or command output. These are particularly useful for templating solutions where a line containing a keyword gets replaced with text from elsewhere. In the next chapter, you'll learn how to implement control structures using branch commands.

Exercises

a) Replace third to fifth lines of input file `addr.txt` with second to fourth lines from file `para.txt`

```
$ sed ##### add your solution here
Hello World
How are you
Start working on that
project you always wanted
to, do not let it end
You are funny
```

b) Add one line from `hex.txt` after every two lines of `copyright.txt`

```
$ sed ##### add your solution here
bla bla 2015 bla
blah 2018 blah
start address: 0xA0, func1 address: 0xA0
bla bla bla
```

```
copyright: 2019
end address: 0xFF, func2 address: 0xB0
```

c) For every line of the input file `hex.txt` , insert `---` before the line and add one line from `replace.txt` after the line as shown below.

```
$ sed ##### add your solution here
---
start address: 0xA0, func1 address: 0xA0
0xA0 0x5000
---
end address: 0xFF, func2 address: 0xB0
0xB0 0x6000
```

d) Insert the contents of `hex.txt` file before a line matching `0x6000` of the input file `replace.txt` .

```
$ sed ##### add your solution here
0xA0 0x5000
start address: 0xA0, func1 address: 0xA0
end address: 0xFF, func2 address: 0xB0
0xB0 0x6000
0xFF 0x7000
```

Control structures

`sed` supports two types of branching commands that helps to construct control structures. These commands (and other advanced features not discussed in this book) allow you to emulate a wide range of features that are common in programming languages. This chapter will show basic examples and you'll find some more use cases in a later chapter.

See also catonmat: A proof that Unix utility sed is Turing complete

Branch commands

Command	Description
`b label`	unconditionally branch to specified `label`
`b`	skip rest of the commands and start next cycle
`t label`	branch to specified `label` on successful substitution
`t`	on successful substitution, skip rest of the commands and start next cycle
`T label`	branch to specified `label` if substitution fails
`T`	if substitution fails, skip rest of the commands and start next cycle

A label is specified by prefixing a command with `:label` where `label` is the name given to be referenced elsewhere with branching commands. Note that for `t` command, any successful substitution since the last input line was read or a conditional branch will count. So, you could have few failed substitutions and a single successful substitution in any order and the branch will be taken. Similarly, `T` command will branch only if there has been no successful substitution since the last input line was read or a conditional branch.

if-then-else

The branching commands can be used to construct control structures like if-then-else. For example, consider an input file containing numbers in a single column and the task required is to change positive numbers to negative and vice versa. If the line starts with `-` character, you need to delete it and process next input line. Else, you need to insert `-` at start of line to convert positive numbers to negative. Both `b` and `t` commands can be used here as shown below.

```
$ cat nums.txt
3.14
-20000
-51
4567

$ # empty REGEXP section will reuse last REGEXP match, in this case /^-/
$ # also note the use of ; after {} command grouping
$ # 2nd substitute command will execute only if line doesn't start with -
$ sed '/^-/{s///; b}; s/^/-/' nums.txt
```

```
-3.14
20000
51
-4567

$ # t command will come into play if the 1st substitute command succeeds
$ # and thus skip the 2nd substitute command
$ sed '/^-/ s///; t; s/^/-/' nums.txt
-3.14
20000
51
-4567
```

The `T` command will branch only if there has been no successful substitution since the last input was read or conditional branch. Rephrased it another way, the commands after the `T` branch will be executed only if there has been at least one successful substitution.

```
$ # 2nd substitution will work only if 1st one succeeds
$ # same as: sed '/o/{s//-/g; s/d/*/g}'
$ printf 'good\nbad\n' | sed 's/o/-/g; T; s/d/*/g'
g--*
bad

$ # append will work if any of the substitution succeeds
$ printf 'good\nbad\nneed\n' | sed 's/o/-/g; s/a/%/g; T; a ----'
g--d
----
b%d
----
need
```

loop

Without labels, branching commands will skip rest of the commands and then start processing the next line from input. By marking a command location with a label, you can branch to that particular location when required. In this case, you'll still be processing the current pattern space.

The below example replaces all consecutive digit characters from start of line with `*` character. `:a` marks the substitute command with label named `a` and `ta` would branch to label `a` if the substitute command succeeds. Effectively, you get a **looping** mechanism to replace the current line as long as the substitute condition is satisfied.

```
$ # same as: perl -pe 's/\G\d/*/g'
$ # first, * is matched 0 times followed by the digit 1
$ # next, * is matched 1 times followed by the digit 2
$ # then, * is matched 2 times followed by the digit 3
$ # and so on until the space character breaks the loop
$ echo '12345 hello42' | sed -E ':a s/^(\**)[0-9]/\1*/; ta'
***** hello42
```

```
$ # here, the x character breaks the loop
$ echo '123x45 hello42' | sed -E ':a s/^(\**)[0-9]/\1*/; ta'
***x45 hello42
$ # no change as the input didn't start with a number
$ echo 'hi 12345 hello42' | sed -E ':a s/^(\**)[0-9]/\1*/; ta'
hi 12345 hello42
```

For debugging purposes, which also helps beginners to understand this command better, unroll the loop and test the command. For the above example, try `sed -E 's/^(\**)[0-9]/\1*/'` followed by `sed -E 's/^(\**)[0-9]/\1*/; s//\1*/'` and so on.

Space between `:` and label name is optional. Similarly, space between branch command and target label is optional.

Here's an example for field processing. `awk` and `perl` are better suited for field processing, but in some cases `sed` might be convenient because rest of the text processing is already in `sed` and so on.

```
$ # replace space with underscore only in 2nd column
$ # [^,]*, captures first column delimited by comma character
$ # [^ ,]* matches non-space and non-comma characters
$ # end of line or another comma will break the loop
$ echo 'he be me,1 2 3 4,nice slice' | sed -E ':b s/^([^,]*,[^ ,]*) /\1_/; tb'
he be me,1_2_3_4,nice slice
```

The looping construct also helps to emulate certain advanced regular expression features not available in `sed` like lookarounds (see stackoverflow: regex faq).

```
$ # replace empty fields with NA
$ # simple replacement won't work for ,,, case
$ echo '1,,,two,,3' | sed 's/,,/,NA,/g'
1,NA,,two,NA,3
$ # looping to the rescue
$ echo '1,,,two,,3' | sed -E ':c s/,,/,NA,/g; tc'
1,NA,NA,two,NA,3
```

The below example has similar solution to previous example, but the problem statement is different and cannot be solved using lookarounds. Here, the act of performing substitution results in an output string that will again match the search pattern.

```
$ # deleting 'fin' results in 'cofing' which can again match 'fin'
$ echo 'coffining' | sed 's/fin//'
cofing
$ # add more s commands if number of times to substitute is known
$ echo 'coffining' | sed 's/fin//; s///'
cog
$ # use loop when it is unknown
$ echo 'coffining' | sed ':d s/fin//; td'
cog
```

Cheatsheet and summary

Note	Description
`b label`	unconditionally branch to specified `label`
`b`	skip rest of the commands and start next cycle
`t label`	branch to specified `label` on successful substitution
`t`	on successful substitution, skip rest of the commands and start next cycle
`T label`	branch to specified `label` if substitution fails
`T`	if substitution fails, skip rest of the commands and start next cycle

This chapter introduced branching commands that can be used to emulate programming features like if-else and loops. These are handy for certain cases, especially when combined with filtering features of `sed`. Speaking of filtering features, the next chapter will focus entirely on using address range for various use cases.

Exercises

a) Using the input file `para.txt`, create a file named `markers.txt` with all lines that contain `start` or `end` (matched case insensitively) and a file named `rest.txt` with rest of the lines.

```
$ sed ##### add your solution here
$ cat markers.txt
good start
Start working on that
to, do not let it end
start and try to
finish the End
$ cat rest.txt
project you always wanted
hi there
bye
```

b) For the input file `addr.txt`:

- if line contains `e`, surround all consecutive repeated characters with `{}` as well as uppercase those characters
- if line doesn't contain `e` but contains `u`, surround all uppercase letters in that line with `[]`

```
$ # note that H in second line and Y in last line isn't modified
$ sed ##### add your solution here
He{LL}o World
How are you
This game is g{OO}d
[T]oday is sunny
12345
You are fu{NN}y
```

c) The given sample strings below has multiple fields separated by a space. The first field has numbers separated by `-` character. Surround these numbers in first field with `[]`

```
$ echo '123-87-593 42-3 foo' | sed ##### add your solution here
[123]-[87]-[593] 42-3 foo

$ echo '53783-0913 hi 3 4-2' | sed ##### add your solution here
[53783]-[0913] hi 3 4-2
```

d) Convert the contents of `headers.txt` such that it matches the content of `anchors.txt` . The input file `headers.txt` contains one header per line, starting with one or more `#` character followed by a space character and then followed by the heading. You have to convert this heading into anchor tag as shown by the contents of `anchors.txt` .

```
$ cat headers.txt
# Regular Expressions
## Subexpression calls
## The dot meta character
$ cat anchors.txt
# <a name="regular-expressions"></a>Regular Expressions
## <a name="subexpression-calls"></a>Subexpression calls
## <a name="the-dot-meta-character"></a>The dot meta character

$ sed ##### add your solution here headers.txt > out.txt
$ diff -s out.txt anchors.txt
Files out.txt and anchors.txt are identical
```

Processing lines bounded by distinct markers

Address range was already introduced in an earlier chapter. This chapter will cover a wide variety of use cases where you need to process a group of lines defined by a starting and a ending pattern. For some examples, other text processing commands will also be used to construct a simpler one-liner compared to a complex `sed` only solution.

Uniform markers

This section will cover cases where the input file will always contain the same number of starting and ending patterns and arranged in alternating fashion. For example, there cannot be two starting patterns appearing without an ending pattern between them and vice versa. Lines of text inside and between such groups are optional.

The sample file shown below will be used to illustrate examples in this section. For simplicity, assume that the starting pattern is marked by `start` and the ending pattern by `end` . They have also been given group numbers to make it easier to visualize the transformation between input and output for the commands discussed in this section.

```
$ cat uniform.txt
mango
icecream
--start 1--
1234
6789
**end 1**
how are you
have a nice day
--start 2--
a
b
c
**end 2**
par,far,mar,tar
```

Case 1: Processing all the group of lines based on the distinct markers, including the lines matched by markers themselves. For simplicity, the below command will just print all such lines. This use case was already covered in Address range section as well.

```
$ sed -n '/start/,/end/p' uniform.txt
--start 1--
1234
6789
**end 1**
--start 2--
a
b
c
**end 2**
```

Case 2: Processing all the group of lines but excluding the lines matched by markers themselves.

```
$ # recall that empty REGEXP will reuse last matched REGEXP
$ sed -n '/start/,/end/{//! s/^/* /p}' uniform.txt
* 1234
* 6789
* a
* b
* c
```

Case 3: Processing all the group of lines but excluding the ending marker.

```
$ sed -n '/start/,/end/{/end/!p}' uniform.txt
--start 1--
1234
6789
--start 2--
a
b
c
```

Case 4: Processing all the group of lines but excluding the starting marker.

```
$ sed -n '/start/,/end/{/start/!p}' uniform.txt
1234
6789
**end 1**
a
b
c
**end 2**
```

Case 5: Processing all input lines except the group of lines bound by the markers.

```
$ sed '/start/,/end/d; s/$/./' uniform.txt
mango.
icecream.
how are you.
have a nice day.
par,far,mar,tar.
```

Case 6 Processing all input lines except the group of lines between the markers.

```
$ sed '/start/,/end/{//!d}' uniform.txt
mango
icecream
--start 1--
**end 1**
how are you
have a nice day
--start 2--
**end 2**
par,far,mar,tar
```

Case 7: Similar to case 6, but include the starting marker.

```
$ sed '/start/,/end/{/start/!d}' uniform.txt
mango
icecream
--start 1--
how are you
have a nice day
--start 2--
par,far,mar,tar
```

Case 8: Similar to case 6, but include the ending marker.

```
$ sed '/start/,/end/{/end/!d}' uniform.txt
mango
icecream
**end 1**
how are you
have a nice day
**end 2**
par,far,mar,tar
```

Extracting first or last group

The same sample input file from the previous section will be used for this section's examples as well. The task is to extract only the first or the very last group of lines defined by the markers.

To get the first block, simply apply `q` command when the ending mark is matched.

```
$ sed -n '/start/,/end/{p; /end/q}' uniform.txt
--start 1--
1234
6789
**end 1**

$ # use other tricks discussed in previous section as needed
$ sed -n '/start/,/end/{//!p; /end/q}' uniform.txt
1234
6789
```

To get the last block, reverse the input linewise, change the order of address range, get the first block, and then reverse linewise again.

```
$ tac uniform.txt | sed -n '/end/,/start/{p; /start/q}' | tac
--start 2--
a
b
c
**end 2**
```

Broken groups

Sometimes, the starting and ending markers aren't always present uniformly in pairs. For example, consider a log file which can have multiple warning messages followed by an error message as shown below.

```
$ cat log.txt
foo baz 123
--> warning 1
a,b,c,d
42
--> warning 2
x,y,z
--> warning 3
4,3,1
==> error 1
hi bye
```

Considering error lines as the ending marker, the starting marker might be one of two possibilities. Either get all the warning messages or get only the last warning message that occurs before the error.

```
$ sed -n '/warning/,/error/p' log.txt
--> warning 1
a,b,c,d
42
--> warning 2
x,y,z
--> warning 3
4,3,1
==> error 1

$ tac log.txt | sed -n '/error/,/warning/p' | tac
--> warning 3
4,3,1
==> error 1
```

If both the starting and ending markers can occur multiple times, then learn_gnuawk: broken blocks or learnbyexample perl: broken blocks would suit better than trying to solve with `sed`

Summary

This chapter didn't introduce any new feature, but rather dealt with a variety of use cases that need the address range filter. Some of them required using other commands to make the solution simpler. The next chapter will discuss various gotchas that you may encounter while using `sed` and a few tricks to get better performance. After that, there's another chapter with resource links for further reading. Hope you found `sed` as an interesting and useful

tool to learn. Happy coding!

Exercises

a) For the input file `broken.txt`, print all lines between the markers `top` and `bottom`. The first `sed` command shown below doesn't work because `sed` will match till end of file if second address isn't found.

```
$ cat broken.txt
top
3.14
bottom
top
1234567890
bottom
top
Hi there
Have a nice day
Good bye

$ # wrong output
$ sed -n '/top/,/bottom/ {//!p}' broken.txt
3.14
1234567890
Hi there
Have a nice day
Good bye

$ # expected output
$ ##### add your solution here
3.14
1234567890
```

b) For the input file `addr.txt`, replace the lines occurring between the markers `How` and `12345` with contents of the file `hex.txt`.

```
$ sed ##### add your solution here
Hello World
How are you
start address: 0xA0, func1 address: 0xA0
end address: 0xFF, func2 address: 0xB0
12345
You are funny
```

Gotchas and Tricks

1) Use single quotes to enclose `sed` commands on the command line to avoid potential conflict with shell metacharacters. This case applies when the command doesn't need variable or command substitution.

```
$ # space is a shell metacharacter, hence the error
$ echo 'a sunny day' | sed s/sunny day/cloudy day/
sed: -e expression #1, char 7: unterminated `s' command
$ # shell treats characters inside single quotes literally
$ echo 'a sunny day' | sed 's/sunny day/cloudy evening/'
a cloudy evening
```

2) On the other hand, beginners often do not realize the difference between single and double quotes and expect shell substitutions to work from within single quotes. See wooledge: Quotes and unix.stackexchange: Why does my shell script choke on whitespace or other special characters? for details about various quoting mechanisms.

```
$ # $USER won't get expanded within single quotes
$ echo 'User name: ' | sed 's/$/$USER/'
User name: $USER

$ # use double quotes for such cases
$ echo 'User name: ' | sed "s/$/$USER/"
User name: learnbyexample
```

3) When shell substitution is needed, surrounding entire command with double quotes may lead to issues due to conflict between `sed` and `bash` special characters. So, use double quotes only for the portion of the command where it is required.

```
$ # ! is one of special shell characters within double quotes
$ word='at'
$ printf 'sea\neat\ndrop\n' | sed "/${word}/!d"
printf 'sea\neat\ndrop\n' | sed "/${word}/date -Is"
sed: -e expression #1, char 6: extra characters after command

$ # works correctly when only the required portion is double quoted
$ printf 'sea\neat\ndrop\n' | sed '/'"${word}"'/!d'
eat
```

4) Another gotcha when applying variable or command substitution is the conflict between `sed` metacharacters and the value of the substituted string. See also stackoverflow: Is it possible to escape regex metacharacters reliably with sed and unix.stackexchange: security consideration when using shell substitution.

```
$ # variable being substituted cannot have the delimiter character
$ printf 'home\n' | sed 's/$/: '"$HOME"'/'
sed: -e expression #1, char 8: unknown option to `s'

$ # use a different delimiter that won't conflict with variable value
$ printf 'home\n' | sed 's|$|: '"$HOME"'|'
home: /home/learnbyexample
```

5) You can specify command line options after filename arguments. Useful if you forgot some option(s) and want to edit the previous command from history.

```
$ printf 'boat\nsite\nfoot\n' > temp.txt
$ # no output, as + is not special with default BRE
$ sed -n '/[aeo]+t/p' temp.txt

$ # pressing up arrow will bring up the last command from history
$ # then you can add the option needed at the end of the command
$ sed -n '/[aeo]+t/p' temp.txt -E
boat
foot
```

As a corollary, if a filename starts with `-`, you need to either escape it or use `--` as an option to indicate that no more options will be used. The `--` feature is not unique to `sed` command, it is applicable to many other commands as well and typically used when filenames are obtained from another source or expanded by shell globs such as `*.txt`.

```
$ echo 'hi hello' > -dash.txt
$ sed 's/hi/HI/' -dash.txt
sed: invalid option -- 'd'

$ sed -- 's/hi/HI/' -dash.txt
HI hello

$ # clean up temporary file
$ rm -- -dash.txt
```

6) Your command might not work and/or get weird output if your input file has dos style line endings.

```
$ # substitution doesn't work here because of dos style line ending
$ printf 'hi there\r\ngood day\r\n' | sed -E 's/\w+$/123/'
hi there
good day
$ # matching \r optionally is one way to solve this issue
$ # that way, it'll work for both \r\n and \n line endings
$ printf 'hi there\r\ngood day\r\n' | sed -E 's/\w+(\r?)$/123\1/'
hi 123
good 123

$ # swapping every two columns, works well with \n line ending
$ printf 'good,bad,42,24\n' | sed -E 's/([^,]+),([^,]+)/\2,\1/g'
bad,good,24,42
$ # output gets mangled with \r\n line ending
$ printf 'good,bad,42,24\r\n' | sed -E 's/([^,]+),([^,]+)/\2,\1/g'
,42,good,24
```

I use these `bash` functions (as part of `.bashrc` configuration) to easily switch between dos and unix style line endings. Some Linux distribution may come with these commands installed by default. See also stackoverflow: Why does my tool output overwrite itself and how do I fix it?

```
unix2dos() { sed -i 's/$/\r/' "$@" ; }
dos2unix() { sed -i 's/\r$//' "$@" ; }
```

7) Unlike `grep` , `sed` will not add a newline if last line of input didn't have one.

```
$ # grep added a newline even though 'drop' doesn't end with newline
$ printf 'sea\neat\ndrop' | grep -v 'at'
sea
drop
$ # sed will not do so
$ # note how the prompt appears after 'drop'
$ printf 'sea\neat\ndrop' | sed '/at/d'
sea
drop$
```

8) Use of `-e` option for commands like `a/c/i/r/R` when command grouping is also required.

```
$ # } gets treated as part of argument for append command, hence the error
$ seq 3 | sed '2{s/^/*/; a hi}'
sed: -e expression #1, char 0: unmatched `{'

$ # } now used with -e, but -e is still missing for first half of command
$ seq 3 | sed '2{s/^/*/; a hi' -e '}'
sed: -e expression #1, char 1: unexpected `}'

$ # -e now properly used for both portions of the command
$ seq 3 | sed -e '2{s/^/*/; a hi' -e '}'
1
*2
hi
3
```

9) Longest match wins. See also stackoverflow: Greedy vs Reluctant vs Possessive

```
$ s='food land bark sand band cue combat'
$ # this will always match from first 'foo' to last 'ba'
$ echo "$s" | sed 's/foo.*ba/X/'
Xt
$ # if you need to match from first 'foo' to first 'ba', then
$ # use a tool which supports non-greedy quantifiers
$ echo "$s" | perl -pe 's/foo.*?ba/X/'
Xrk sand band cue combat
```

For certain cases, character class can help in matching only the relevant characters. And in some cases, adding more qualifiers instead of just `.*` can help. See stackoverflow: How to replace everything until the first occurrence for an example.

```
$ echo '{52} apples and {31} mangoes' | sed 's/{.*}/42/g'
42 mangoes
$ echo '{52} apples and {31} mangoes' | sed 's/{[^}]*}/42/g'
42 apples and 42 mangoes
```

10) Beware of empty matches when using the `*` quantifier.

```
$ # * matches zero or more times
$ echo '42,,,,,hello,bye,,,hi' | sed 's/,*/,/g'
,4,2,h,e,l,l,o,b,y,e,h,i,
$ # + matches one or more times
$ echo '42,,,,,hello,bye,,,hi' | sed -E 's/,+/,/g'
42,hello,bye,hi
```

11) BRE vs ERE syntax could get confusing for beginners. Quoting from the manual:

> In GNU sed, the only difference between basic and extended regular expressions is in the behavior of a few special characters: `?` , `+` , parentheses, braces (`{}`), and `|` .

```
$ # no match as + is not special with default BRE
$ echo '52 apples and 31234 mangoes' | sed 's/[0-9]+/[&]/g'
52 apples and 31234 mangoes
$ # so, either use \+ with BRE or use + with ERE
$ echo '52 apples and 31234 mangoes' | sed 's/[0-9]\+/[&]/g'
[52] apples and [31234] mangoes

$ # the reverse is also common, use of escapes when not required
$ echo 'get {} set' | sed 's/\{\}/[]/'
sed: -e expression #1, char 10: Invalid preceding regular expression
$ echo 'get {} set' | sed 's/{}/[]/'
get [] set
```

12) Online tools like regex101 and debuggex can be very useful for beginners to regular expressions, especially for debugging purposes. However, their popularity has lead to users trying out their pattern on these sites and expecting them to work as is for command line tools like `grep` , `sed` and `awk` . The issue arises when features like **non-greedy** and **lookarounds** are used as they wouldn't work with BRE/ERE. See also unix.stackexchange: Why does my regular expression work in X but not in Y?

```
$ echo '1,,,two,,3' | sed -E 's/,\K(?=,)/NA/g'
sed: -e expression #1, char 15: Invalid preceding regular expression
$ echo '1,,,two,,3' | perl -pe 's/,\K(?=,)/NA/g'
1,NA,NA,two,NA,3

$ # \d is not available as character set escape sequence
$ # will match 'd' instead
$ echo '52 apples and 31234 mangoes' | sed -E 's/\d+/[&]/g'
52 apples an[d] 31234 mangoes
$ echo '52 apples and 31234 mangoes' | perl -pe 's/\d+/[$&]/g'
[52] apples and [31234] mangoes
```

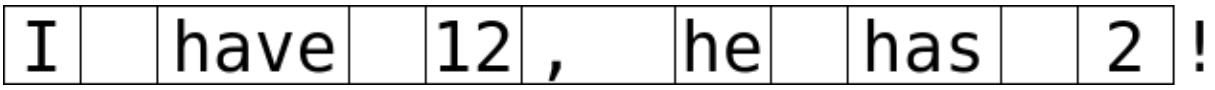
13) If you are facing issues with end of line matching, it is often due to dos-style line ending (discussed earlier in this chapter) or whitespace characters at the end of line.

```
$ # there's no visual clue to indicate whitespace characters at end of line
$ printf 'food bark \n1234 6789\t\n'
```

```
food bark
1234 6789
$ # no match
$ printf 'food bark \n1234 6789\t\n' | sed -E 's/\w+$/xyz/'
food bark
1234 6789

$ # cat command has options to indicate end of line, tabs, etc
$ printf 'food bark \n1234 6789\t\n' | cat -A
food bark $
1234 6789^I$
$ # works now, as whitespace characters are matched too
$ printf 'food bark \n1234 6789\t\n' | sed -E 's/\w+\s*$/xyz/'
food xyz
1234 xyz
```

14) The word boundary `\b` matches both start and end of word locations. Whereas, `\<` and `\>` match exactly the start and end of word locations respectively. This leads to cases where you have to choose which of these word boundaries to use depending on results desired. Consider `I have 12, he has 2!` as sample text, shown below as an image with vertical bars marking the word boundaries. The last character `!` doesn't have end of word boundary as it is not a word character.




```
$ # \b matches both start and end of word boundaries
$ # the first match here used starting boundary of 'I' and 'have'
$ echo 'I have 12, he has 2!' | sed 's/\b..\b/[&]/g'
[I ]have [12][, ][he] has[ 2]!

$ # \< and \> only match the start and end word boundaries respectively
$ echo 'I have 12, he has 2!' | sed 's/\<..\>/[&]/g'
I have [12], [he] has 2!
```

Here's another example to show the difference between the two types of word boundaries.

```
$ # add something to both start/end of word
$ echo 'hi log_42 12b' | sed 's/\b/:/g'
:hi: :log_42: :12b:

$ # add something only at start of word
$ echo 'hi log_42 12b' | sed 's/\</:/g'
:hi :log_42 :12b

$ # add something only at end of word
$ echo 'hi log_42 12b' | sed 's/\>/:/g'
hi: log_42: 12b:
```

15) For some cases, you could simplify and improve readability of a substitution command by adding a filter condition instead of using substitution only.

```
$ # insert 'Error: ' at start of line if the line contains '42'
$ # also, remove all other starting whitespaces for such lines
$ printf '1423\n214\n   425\n' | sed -E 's/^\s*(.*42)/Error: \1/'
Error: 1423
214
Error: 425

$ # simpler and readable
$ # also note that -E is no longer required
$ printf '1423\n214\n   425\n' | sed '/42/ s/^\s*/Error: /'
Error: 1423
214
Error: 425
```

16) Both `1` and `$` will match as an address if input file has only one line of data.

```
$ printf '3.14\nhi\n42\n' | sed '1 s/^/start: /; $ s/$/ :end/'
start: 3.14
hi
42 :end
$ echo '3.14' | sed '1 s/^/start: /; $ s/$/ :end/'
start: 3.14 :end

$ # you could use control structures as a workaround
$ # this will not work for ending address if input has only one line
$ echo '3.14' | sed '1{s/^/start: /; b}; $ s/$/ :end/'
start: 3.14
$ # this will not work for starting address if input has only one line
$ echo '3.14' | sed '${s/$/ :end/; b}; 1 s/^/start: /'
3.14 :end
```

17) `n` and `N` commands will not execute further commands if there's no more input lines to fetch.

```
$ # last line matched the filtering condition
$ # but substitution didn't work for last line as there's no more input
$ printf 'red\nblue\ncredible\n' | sed '/red/{N; s/e.*e/2/}'
r2
credible

$ # $!N will avoid executing N command for last line of input
$ printf 'red\nblue\ncredible\n' | sed '/red/{$!N; s/e.*e/2/}'
r2
cr2
```

18) Changing locale to ASCII (assuming default is not ASCII locale) can give significant speed boost.

```
$ # time shown is best result from multiple runs
$ # speed benefit will vary depending on computing resources, input, etc
$ time sed -nE '/^([a-d][r-z]){3}$/p' /usr/share/dict/words > f1
real    0m0.040s
```

```
$ # LC_ALL=C will give ASCII locale, active only for this command
$ time LC_ALL=C sed -nE '/^([a-d][r-z]){3}$/p' /usr/share/dict/words > f2
real    0m0.016s

$ # check that results are same for both versions of the command
$ diff -s f1 f2
Files f1 and f2 are identical
```

Here's another example.

```
$ time sed -nE '/^([a-z]..)\1$/p' /usr/share/dict/words > f1
real    0m0.082s

$ time LC_ALL=C sed -nE '/^([a-z]..)\1$/p' /usr/share/dict/words > f2
real    0m0.048s

$ # clean up temporary files
$ rm f[12]
```

19) ripgrep (command name `rg`) is primarily used as an alternative to `grep` but also supports search and replace functionality. It has more regular expression features than BRE/ERE, supports unicode, multiline and fixed string matching and generally faster than `sed`. `sed 's/search/replace/g' file` is similar to `rg --passthru -N 'search' -r 'replace' file`. There are plenty of features to recommended learning `rg` even though it supports substitution in limited fashion compared to `sed` (no in-place support, no address filtering, no control structures, etc). See my book on GNU GREP and RIPGREP for more details.

```
$ # same as: sed 's/e/E/g' greeting.txt
$ # --passthru is needed to print lines which didn't match the pattern
$ rg --passthru -N 'e' -r 'E' greeting.txt
Hi thErE
HavE a nicE day

$ # non-greedy quantifier
$ s='food land bark sand band cue combat'
$ echo "$s" | rg --passthru 'foo.*?ba' -r 'X'
Xrk sand band cue combat

$ # Multiline search and replacement
$ printf '42\nHi there\nHave a Nice Day' | rg --passthru -U '(?s)the.*ice' -r ''
42
Hi  Day

$ # easily handle fixed strings, this one replaces [4]* with 2
$ printf '2.3/[4]*6\nfoo\n5.3-[4]*9\n' | rg --passthru -F '[4]*' -r '2'
2.3/26
foo
5.3-29
```

```
$ # unicode support
$ echo 'fox:αλεπού,eagle:αετός' | rg '\p{L}+' -r '($0)'
(fox):(αλεπού),(eagle):(αετός)

$ # -P option enables PCRE2 if you need even more advanced features
$ echo 'car bat cod map' | rg -P '(bat|map)(*SKIP)(*F)|\w+' -r '[$0]'
[car] bat [cod] map
```

Further Reading

- `man sed` and `info sed` and online manual
- Information about various implementations of `sed`
  - sed FAQ, great resource, but last modified *10 March 2003*
  - stackoverflow: BSD/macOS sed vs GNU sed vs the POSIX sed specification
  - unix.stackexchange: Differences between sed on Mac OSX and other standard sed
  - grymoire: sed tutorial — has details on differences between various `sed` versions as well
- Q&A on stackoverflow/stackexchange are good source of learning material, good for practice exercises as well
  - sed Q&A on unix stackexchange
  - sed Q&A on stackoverflow
- Learn Regular Expressions (has information on flavors other than BRE/ERE too)
  - regular-expressions — tutorials and tools
  - rexegg — tutorials, tricks and more
  - stackoverflow: What does this regex mean?
  - online regex tester and debugger — not fully suitable for cli tools, but most of ERE syntax works
- My repo on cli text processing tools
- Related tools
  - rpl — search and replace tool, has interesting options like interactive mode and recursive mode
  - sedsed — Debugger, indenter and HTMLizer for sed scripts
  - xo — composes regular expression match groups
  - sd — simple search and replace, implemented in Rust
- unix.stackexchange: When to use grep, sed, awk, perl, etc

Here's some links for specific topics:

- ASCII reference and locale usage
  - ASCII code table
  - wiki.archlinux: locale
  - shellhacks: Define Locale and Language Settings
- unix.stackexchange: replace multiline string
- stackoverflow: deleting empty lines with optional white spaces
- unix.stackexchange: print only line above the matching line
- stackoverflow: get lines between two patterns only if there is third pattern between them
  - unix.stackexchange: similar example